References

[1] G. Hadjiyiannis, S. Hanono, and S. Devadas. ISDL: An Instruction Set Description Language for Retargetability. In *Proceedings of the* 34^{th} *Design Automation Conference*, pages 299–302, June 1997.

[2] G. Goossens, F. Catthoor, D. Lanneer, and H. De Man. Integration of Signal Processing Systems on Heterogeneous IC Architectures. In *Proceedings of the 6th International Workshop on High-Level Synthesis*, pages 16–26, November 1992.

[3] R. K. Gupta and G. De Micheli. Hardware–Software Cosynthesis for Digital Systems. *IEEE Design & Test of Computers*, pages 29–41, September 1993.

[4] J. Levine, T. Mason, and D. Brown. *lex & yacc*. O'Reilly & Associates, Inc., 1992.

```
~((Main_ORI *,MR) & [1](Main_RTI))
~((Main_ANDI *,CCR) & [1](Main_RTI))
~((Main_ORI *,CCR) & [1](Main_RTI))
// An RTI cannot be the last instruction in a DO loop (at LA)

// The following instructions can never appear immediately before an RTS
//      MOVEC to LA, LC, SSH, SSL, or SP
//      MOVEM to LA, LC, SSH, SSL, or SP
//      MOVEP to LA, LC, SSH, SSL, or SP
//      MOVEC from SSH
//      MOVEM from SSH
//      MOVEP from SSH
~((((((Main_MOVE? *, LA) |
  (Main_MOVE? *, LC))  |
  (Main_MOVE? *, SSH)) |
  (Main_MOVE? *, SSL)) |
  (Main_MOVE? *, SP)) & [1](Main_RTS))
// You cannot repeat an RTS usign REP
~((Main_REP *) & [1](Main_RTS))
// Also, an RTS cannot be the last instruction in a DO loop (at LA)

// A STOP instruction cannot be used in a fast interrupt routine

// An SWI instucttion cannot be used in a fast interrupt routine

// A WAIT instruction cannot be used in a fast interrupt routine
// A WAIT instruction cannot be the last instruction in a DO loop (at LA)

// ------------------------------------------------------------------------

Section Optional
```

```
// A JSSET instruction cannot be repeated with a REP instruction
~((Main_REP *) & [1](Main_JSSET *))
// A JSSET SSH or JSSET SSL cannot follow an instruction that changes SP
~((((((((((((((((Main_DO *) | (Main_ENDDO)) | (Main_JS *)) | (Main_JSI *)) |
   (Main_JSCLR *)) | (Main_JSRI *)) | (Main_ JSR *)) | (Main_JSSET *)) |
   (Main_RTI)) | (Main_RTS)) | (Main_BCHG *,SP)) | (Main_BCLR *,SP)) |
   (Main_BSET *,SP)) | (Main_MOVEC *,SP)) | (Main_MOVEM *,SP)) |
   (Main_MOVEP *,SP)) & [1]((Main_JSSET *,SSH,*) | (Main_JSSET *,SSL,*)))
// A JSSET located at LA, LA-1, or LA-2 of the DO loop cannot specify
// the program controller registers SR, SP, SSH, SSL, LA, or LC as
// it's target
// A JSSET instruction used within a DO loop cannot specify the
// loop address (LA) as it's target

// A MOVEC SSH, SSH instruction is illegal and cannot be used
~(Main_MOVEC SSH,SSH)

// A MOVE? instruction which specifies SP as the destination operand
// cannot be used immediately before a MOVEC, MOVEM, or MOVEP instruction
// which specifies SSH or SSL as the source operand.
~((Main_MOVE? *,SP) & [1]((Main_MOVE? SSH,*) | (Main_MOVE? SSL,*)))

// A REP instruction cannot repeat the following:
// DO, J, JI, JCLR, JMP, JSET, JS, JSI, JSCLR, JSR, JSRI, JSSET, REP,
// RTI, RTS, STOP, SWI, WAIT, ENDDO
~((Main_REP *) & [1](((((((((((((((((((Main_DO *) | (Main_J *)) |
                  (Main_JI *)) | (Main_JCLR *)) | (Main_JMP *)) |
                  (Main_JMPI *)) | (Main_JSET *)) | (Main_JS *)) |
                  (Main_JSI *)) | (Main_JSCLR *)) | (Main_JSR *)) |
                  (Main_JSRI *)) | (Main_JSSET *)) | (Main_REP *)) |
                  (Main_RTI)) | (Main_RTS)) | (Main_STOP)) | (Main_SWI)) |
                  (Main_WAIT)) | (Main_ENDDO)))
// Also, the REP instruction cannot be the last instruction in a DO loop

// A RESET cannot be the last instruction in a DO loop (at LA).

// The following instructions can never appear immediately before an RTI
//      MOVEC to LA, LC, SSH, SSL, or SP
//      MOVEM to LA, LC, SSH, SSL, or SP
//      MOVEP to LA, LC, SSH, SSL, or SP
//      MOVEC from SSH
//      MOVEM from SSH
//      MOVEP from SSH
//      ANDI *,MR
//      ORI *,MR
//      ANDI *,CCR
//      ORI *,CCR
~((((((Main_MOVE? *, LA) |
  (Main_MOVE? *, LC))  |
  (Main_MOVE? *, SSH)) |
  (Main_MOVE? *, SSL)) |
  (Main_MOVE? *, SP)) & [1](Main_RTI))
~((Main_MOVE? SSH, *) & [1](Main_RTI))
~((Main_ANDI *,MR) & [1](Main_RTI))
```

```
// A JCLR located at LA, LA-1, or LA-2 of the DO loop cannot specify
// the program controller registers SR, SP, SSH, SSL, LA, or LC as
// it's target

// A JMP instruction cannot be repeated with a REP instruction
~((Main_REP *) & [1](Main_JMP *))
~((Main_REP *) & [1](Main_JMPI *))
// A JMP instruction used within a DO loop cannot begin at the address
// LA within that DO loop

// A JS C_CODE instruction cannot be repeated with a REP instruction
~((Main_REP *) & [1](Main_JS *))
~((Main_REP *) & [1](Main_JSI *))
// A JS C_CODE instruction used within a DO loop cannot begin at the address
// LA within that DO loop
// A JS C_CODE instruction used within a DO loop cannot specify the
// loop address (LA) as it's target

// A JSCLR instruction cannot be repeated with a REP instruction
~((Main_REP *) & [1](Main_JSCLR *))
// A JSCRL SSH or JSCLR SSL cannot follow an instruction that changes SP
~(((((((((((((((((Main_DO *) | (Main_ENDDO)) | (Main_JS *)) | (Main_JSI *)) |
   (Main_JSCLR *)) | (Main_JSRI *)) | (Main_ JSR *)) | (Main_JSSET *)) |
   (Main_RTI)) | (Main_RTS)) | (Main_BCHG *,SP)) | (Main_BCLR *,SP)) |
   (Main_BSET *,SP)) | (Main_MOVEC *,SP)) | (Main_MOVEM *,SP)) |
   (Main_MOVEP *,SP)) & [1]((Main_JSCLR *,SSH,*) | (Main_JSCLR *,SSL,*)))
// A JSCLR located at LA, LA-1, or LA-2 of the DO loop cannot specify
// the program controller registers SR, SP, SSH, SSL, LA, or LC as
// it's target
// A JSCLR instruction used within a DO loop cannot specify the
// loop address (LA) as it's target

// A JSET instruction cannot be repeated with a REP instruction
~((Main_REP *) & [1](Main_JSET *))
// A JSET SSH or JSET SSL cannot follow an instruction that changes SP
~(((((((((((((((((Main_DO *) | (Main_ENDDO)) | (Main_JS *)) | (Main_JSI *)) |
   (Main_JSCLR *)) | (Main_JSRI *)) | (Main_ JSR *)) | (Main_JSSET *)) |
   (Main_RTI)) | (Main_RTS)) | (Main_BCHG *,SP)) | (Main_BCLR *,SP)) |
   (Main_BSET *,SP)) | (Main_MOVEC *,SP)) | (Main_MOVEM *,SP)) |
   (Main_MOVEP *,SP)) & [1]((Main_JSET *,SSH,*) | (Main_JSET *,SSL,*)))
// A JSET located at LA, LA-1, or LA-2 of the DO loop cannot specify
// the program controller registers SR, SP, SSH, SSL, LA, or LC as
// it's target
// A JSET instruction used within a DO loop cannot specify the
// loop address (LA) as it's target

// A JSR instruction cannot be repeated with a REP instruction
~((Main_REP *) & [1](Main_JSR *))
~((Main_REP *) & [1](Main_JSRI *))
// A JSR instruction used within a DO loop cannot begin at the address
// LA within that DO loop
// A JSR instruction used within a DO loop cannot specify the
// loop address (LA) as it's target
```

```
// At LA-2, LA-1, and LA
//      DO
//      MOVEC to LA, LC, SSH, SSL, or SP
//      MOVEM to LA, LC, SSH, SSL, or SP
//      MOVEP to LA, LC, SSH, SSL, or SP
//      MOVEC from SSH
//      MOVEM from SSH
//      MOVEP from SSH
//      ANDI *,MR
//      ORI *,MR
//      Two word instructions which read LC, SP, or SSL
// At LA-1
//      single word instructions (except REP) which read LC, SP, or SSL
//      JCLR
//      JSET
//      Two word JMP
//      Two word J C_CODE
// At LA
//      Any two word instruction
//      J C_CODE, JCLR, JSET, JMP, JS C_CODE, JSR, REP, RESET, RTI,
//      RTS, STOP, WAIT, ENDDO

// ENDDO timing is just as demanding as DO
// The following instructions can never appear immediately before an ENDDO
//      MOVEC to LA, LC, SSH, SSL, or SP
//      MOVEM to LA, LC, SSH, SSL, or SP
//      MOVEP to LA, LC, SSH, SSL, or SP
//      MOVEC from SSH
//      MOVEM from SSH
//      MOVEP from SSH
//      ANDI *,MR
//      ORI *,MR
~((((((Main_MOVE? *, LA) |
  (Main_MOVE? *, LC))  |
  (Main_MOVE? *, SSH)) |
  (Main_MOVE? *, SSL)) |
  (Main_MOVE? *, SP)) & [1](Main_ENDDO))
~((Main_MOVE? SSH, *) & [1](Main_ENDDO))
~((Main_ANDI *,MR) & [1](Main_ENDDO))
~((Main_ORI *,MR) & [1](Main_ENDDO))
// ENDDO is also illegal as the last instruction in a DO loop (at LA)

// A J C_CODE instruction cannot be repeated with a REP instruction
~((Main_REP *) & [1](Main_J *))
~((Main_REP *) & [1](Main_JI *))

// A JCLR instruction cannot be repeated with a REP instruction
~((Main_REP *) & [1](Main_JCLR *))
// A JCRL SSH or JCLR SSL cannot follow an instruction that changes SP
~((((((((((((((((Main_DO *) | (Main_ENDDO)) | (Main_JS *)) | (Main_JSI *)) |
   (Main_JSCLR *)) | (Main_JSRI *)) | (Main_ JSR *)) | (Main_JSSET *)) |
   (Main_RTI)) | (Main_RTS)) | (Main_BCHG *,SP)) | (Main_BCLR *,SP)) |
   (Main_BSET *,SP)) | (Main_MOVEC *,SP)) | (Main_MOVEM *,SP)) |
   (Main_MOVEP *,SP)) & [1]((Main_JCLR *,SSH,*) | [1](Main_JCLR *,SSL,*)))
```

```
~(Main_CMP @[1], @[1])
~(Main_CMPM @[1], @[1])
~(Main_SUBL @[1], @[1])
~(Main_SUBR @[1], @[1])

// Where you do parallel moves to both X and Y mem banks, make sure they use
// opposite banks in their addressing modes.
~(DBM_Move *R[0123]*R[0123]*)
~(DBM_Move *R[4567]*R[4567]*)

// The ANDI IMM, MR instruction cannot be used immediately before an ENDDO
// or RTI instruction and cannot be one of the last three instructions in a
// DO loop (at LA-2, LA-1, or LA)
// ***** we have no way of describing the latter part of this constraint for
// the moment
~((Main_ANDI *, MR) & [1]((Main_ENDDO) | (Main_RTI)))

// ANDI IMM, CCR cannot be used immediately before an RTI
~((Main_ANDI *, CCR) & [1](Main_RTI))

// Do is very picky as to what can come before or after it simply
// because of all the control flow that has to be taken care of.

// Instructions that cannot be performed immediately before a DO
// MOVEC to LA, LC, SSH, SSL, or SP
// MOVEM to LA, LC, SSH, SSL, or SP
// MOVEP to LA, LC, SSH, SSL, or SP
// MOVEC from SSH
// MOVEM from SSH
// MOVEP from SSH
~((((((Main_MOVE? *, LA) |
  (Main_MOVE? *, LC))  |
  (Main_MOVE? *, SSH)) |
  (Main_MOVE? *, SSL)) |
  (Main_MOVE? *, SP)) & [1](Main_DO *))
~((Main_MOVE? SSH, *) & [1](Main_DO *))

// You cannot repeat a DO usign REP
~((Main_REP *) & [1](Main_DO *))

// This is also illegal
~(Main_DO SSH,*)

// The following are also illegal but we have no way of describing the
// fact so far

// The following instructions are not valid when the L_F bit is set
// JSR to LA
// JS C_CODE to LA
// JSCLR to LA
// JSSET to LA

// The following instructions cannot begin at the indicated positions
// near the end of a DO loop
```

```
            { NOP(); }
            {}
            { Cycle = 0; Size = 0; Stall = 0; }
            { Latency=1; Usage=1; }

// ------------------------------------------------------------------------
// Any restrictions (in the form of rules) of which instructions go together
// and which do not.

Section Constraints

// Moves cannot be done at the same time as some operations because of a
// bitfield conflict on the DBM fields:
// Note that the list of instructions could be made shorter using wildcards
// at the expense of convenience though.

~( (DBM_* *) &
   ((((((((((((((((((((((((((((((((Main_ANDI *)       |
    (Main_BCHG *))       |
    (Main_BCLR *))       |
    (Main_BSET *))       |
    (Main_BTST *))       |
    (Main_DIV *))        |
    (Main_DO *))         |
    (Main_ENDDO))        |
    (Main_ILLEGAL))      |
    (Main_J *))          |
    (Main_JI *))         |
    (Main_JCLR *))       |
    (Main_JMP *))        |
    (Main_JMPI *))       |
    (Main_JS *))         |
    (Main_JSI *))        |
    (Main_JSCLR *))      |
    (Main_JSET *))       |
    (Main_JSR* *))       |
    (Main_JSSET *))      |
    (Main_LUA *))        |
    (Main_MOVEC *))      |
    (Main_MOVEM *))      |
    (Main_MOVEP *))      |
    (Main_NORM *))       |
    (Main_ORI *))        |
    (Main_REP *))        |
    (Main_RESET))        |
    (Main_RTI))          |
    (Main_RTS))          |
    (Main_STOP))         |
    (Main_SWI))          |
    (Main_T *))          |
    (Main_WAIT)))

// You cannot do things like Main_ADD A, A etc
~(Main_ADD @[1], @[1])
```

```
        { if (PLIMIT) { L_F <- 1; }; EA; }
        { Cycle = EA; Size = EA; Stall = 0; }
        { Latency=1; Usage=1; }
DBM_Move LMEM, EA, LREG      { DBM.OP = 0x40 | LREG & 0x3 |
                               ((LREG & 0x4) << 1);
                               DBM.MODE = 0xC0 | EA; }
        { LREG <- LMEMS[EA]; }
        { if (PLIMIT) { L_F <- 1; }; EA; }
        { Cycle = EA; Size = EA; Stall = 0; }
        { Latency=1; Usage=1; }
DBM_Move LREG, LMEM, EA      { DBM.OP = 0x40 | LREG & 0x3 |
                               ((LREG & 0x4) << 1);
                               DBM.MODE = 0x40 | EA; }
        { LMEMS[EA] <- LREG; }
        { if (PLIMIT) { L_F <- 1; }; EA; }
        { Cycle = EA; Size = EA; Stall = 0; }
        { Latency=1; Usage=1; }
DBM_MoveP XMEM, DXEA, XA, YMEM, DYEA, YA
                             { DBM.OP = 0xC0 | YA | (XA << 2) |
                               ((DYEA & 0xC) << 2);
                               DBM.MODE = 0x80 | DXEA |
                               ((DYEA & 0x3) << 5); }
        { XA <- XMEMS[DXEA]; YA <- YMEMS[DYEA]; }
        { if (PLIMIT) { L_F <- 1; }; DXEA; DYEA; }
        { Cycle = DXEA + DYEA; Size = DXEA + DYEA; Stall = 0; }
        { Latency=1; Usage=1; }
DBM_MoveP XMEM, DXEA, XA, YA, YMEM, DYEA
                             { DBM.OP = 0x80 | YA | (XA << 2) |
                               ((DYEA & 0xC) << 2);
                               DBM.MODE = 0x80 | DXEA |
                               ((DYEA & 0x3) << 5); }
        { XA <- XMEMS[DXEA]; YMEMS[DYEA] <- YA; }
        { if (PLIMIT) { L_F <- 1; }; DXEA; DYEA; }
        { Cycle = DXEA + DYEA; Size = DXEA + DYEA; Stall = 0; }
        { Latency=1; Usage=1; }
DBM_MoveP XA, XMEM, DXEA, YMEM, DYEA, YA
                             { DBM.OP = 0xC0 | YA | (XA << 2) |
                               ((DYEA & 0xC) << 2);
                               DBM.MODE = 0x00 | DXEA |
                               ((DYEA & 0x3) << 5); }
        { XMEMS[DXEA] <- XA; YA <- YMEMS[DYEA]; }
        { if (PLIMIT) { L_F <- 1; }; DXEA; DYEA; }
        { Cycle = DXEA + DYEA; Size = DXEA + DYEA; Stall = 0; }
        { Latency=1; Usage=1; }
DBM_MoveP XA, XMEM, DXEA, YA, YMEM, DYEA
                             { DBM.OP = 0x80 | YA | (XA << 2) |
                               ((DYEA & 0xC) << 2);
                               DBM.MODE = 0x00 | DXEA |
                               ((DYEA & 0x3) << 5); }
        { XMEMS[DXEA] <- XA; YMEMS[DYEA] <- YA; }
        { if (PLIMIT) { L_F <- 1; }; DXEA; DYEA; }
        { Cycle = DXEA + DYEA; Size = DXEA + DYEA; Stall = 0; }
        { Latency=1; Usage=1; }
DBM_NOP                      { DBM.OP = 0x00; DBM.MODE = 0x00; }
```

```
        { if (PLIMIT) { L_F <- 1; }; EA; }
        { Cycle = EA; Size = EA; Stall = 0; }
        { Latency=1; Usage=1; }
DBM_MoveP LIMMD, XA, ACCR, Y_R
                          { DBM.OP = 0x10 | Y_R | (ACCR << 1) |
                            (XA << 2); DBM.MODE = 0xB4;
                            Additional(0, Split.DATA = LIMMD;); }
        { XA <- LIMMD; Y[Y_R] <- ACCR[1]; }
        { if (PLIMIT) { L_F <- 1; }; }
        { Cycle = 2; Size = 1; Stall = 0; }
        { Latency=1; Usage=1; }
DBM_MoveP A_D, XMEM, EA, X0_R, A_D1
                          { DBM.OP = 0x08; DBM.MODE = EA; }
        { XMEMS[EA] <- AREG; }
        { if (PLIMIT) { L_F <- 1; }; EA; }
        { Cycle = EA; Size = EA; Stall = 0; }
        { Latency=1; Usage=1; }
DBM_MoveP B_D, XMEM, EA, X0_R, B_D1
                          { DBM.OP = 0x09; DBM.MODE = EA; }
        { XMEMS[EA] <- BREG; BREG[1] <- X0_R; }
        { if (PLIMIT) { L_F <- 1; }; EA; }
        { Cycle = EA; Size = EA; Stall = 0; }
        { Latency=1; Usage=1; }
DBM_MoveP X_R, ACC, YMEM, EA, YA
                          { DBM.OP = 0x10 | YA | (X_R << 2) |
                            (ACC << 3); DBM.MODE = 0x84 | EA; }
        { ACC <- X[X_R]; YA <- YMEMS[EA]; }
        { if (PLIMIT) { L_F <- 1; }; EA; }
        { Cycle = EA; Size = EA; Stall = 0; }
        { Latency=1; Usage=1; }
DBM_MoveP X_R, ACC, YA, YMEM, EA
                          { DBM.OP = 0x10 | YA | (X_R << 2) |
                            (ACC << 3); DBM.MODE = 0x04 | EA; }
        { ACC <- X[X_R]; YMEMS[EA] <- YA; }
        { if (PLIMIT) { L_F <- 1; }; EA; }
        { Cycle = EA; Size = EA; Stall = 0; }
        { Latency=1; Usage=1; }
DBM_MoveP X_R, ACC, LIMMD, YA
                          { DBM.OP = 0x10 | YA | (X_R << 2) |
                            (ACC << 3); DBM.MODE = 0xF4;
                            Additional(0, Split.DATA = LIMMD;); }
        { ACC <- X[X_R]; YA <- LIMMD; }
        { if (PLIMIT) { L_F <- 1; }; }
        { Cycle = 2; Size = 1; Stall = 0; }
        { Latency=1; Usage=1; }
DBM_MoveP YMEM, EA, A_D, A_D1, Y0_R
                          { DBM.OP = 0x08; DBM.MODE = 0x80 | EA; }
        { AREG <- YMEMS[EA]; Y[0] <- AREG; }
        { if (PLIMIT) { L_F <- 1; }; EA; }
        { Cycle = EA; Size = EA; Stall = 0; }
        { Latency=1; Usage=1; }
DBM_MoveP YMEM, EA, B_D, B_D1, Y0_R
                          { DBM.OP = 0x09; DBM.MODE = 0x80 | EA; }
        { BREG <- YMEMS[EA]; Y[0] <- BREG; }
```

```
Field DBM:
    DBM_NULL                    DEFINE_NULL_OP
    DBM_Move SIMMD, LAGREGS     { DBM.OP = 0x20 | LAGREGS;
                                  DBM.MODE = SIMMD & 0xFF; }
            { LAGREGS <- SIMMD; }
            {}
            { Cycle = 0; Size = 0; Stall = 0; }
            { Latency=1; Usage=1; }
    DBM_Move LAGREGS, LAGREGS2  { DBM.OP = 0x20 | (LAGREGS >> 3) & 0x3;
                                  DBM.MODE = LAGREGS2 | ((LAGREGS << 5)
                                  & 0xE0); }
            { LAGREGS <- LAGREGS2; }
            { if (PLIMIT) { L_F <- 1; }; }
            { Cycle = 0; Size = 0; Stall = 0; }
            { Latency=1; Usage=1; }
    DBM_Move SEA                { DBM.OP = 0x20; DBM.MODE = 0x40 | SEA; }
            { SEA; }
            {}
            { Cycle = SEA; Size = SEA; Stall = 0; }
            { Latency=1; Usage=1; }
    DBM_Move MEMS, EA, LAGREGS  { DBM.OP = 0x40 | (MEMS << 3) |
                                  (LAGREGS & 0x07) | ((LAGREGS & 0x18) << 1);
                                  DBM.MODE = 0xC0 | EA; }
            { LAGREGS <- MEMS[EA]; }
            { if (PLIMIT) { L_F <- 1; }; EA; }
            { Cycle = EA; Size = EA; Stall = 0; }
            { Latency=1; Usage=1; }
    DBM_Move LAGREGS, MEMS, EA  { DBM.OP = 0x40 | (MEMS << 3) |
                                  (LAGREGS & 0x07) | ((LAGREGS & 0x18) << 1);
                                  DBM.MODE = 0x40 | EA; }
            { MEMS[EA] <- LAGREGS; }
            { if (PLIMIT) { L_F <- 1; }; EA; }
            { Cycle = EA; Size = EA; Stall = 0; }
            { Latency=1; Usage=1; }
    DBM_Move LIMMD, LAGREGS       { DBM.OP = 0x40 | (LAGREGS & 0x07) |
                                  ((LAGREGS & 0x18) << 1);
                                  DBM.MODE = 0xC0 | 0x34;
                                  Additional(0, Split.DATA = LIMMD;); }
            { LAGREGS <- LIMMD; }
            { if (PLIMIT) { L_F <- 1; }; }
            { Cycle = 2; Size = 1; Stall = 0; }
            { Latency=1; Usage=1; }
    DBM_MoveP XMEM, EA, XA, ACCR, Y_R
                                { DBM.OP = 0x10 | Y_R | (ACCR << 1) |
                                  (XA << 2); DBM.MODE = 0x80 | EA; }
            { XA <- XMEMS[EA]; Y[Y_R] <- ACCR[1]; }
            { if (PLIMIT) { L_F <- 1; }; EA; }
            { Cycle = EA; Size = EA; Stall = 0; }
            { Latency=1; Usage=1; }
    DBM_MoveP XA, XMEM, EA, ACCR, Y_R
                                { DBM.OP = 0x10 | Y_R | (ACCR << 1) |
                                  (XA << 2); DBM.MODE = 0x00 | EA; }
            { XMEMS[EA] <- XA; Y[Y_R] <- ACCR[1]; }
```

```
            { Cycle = 2 + DBM; Size = 1 + DBM; Stall = 0; }
            { Latency=1; Usage=1; }
    Main_SUBL ACC, ACC2         { Main.OP = 0x16 | (ACC2 << 3); }
            { ACC2 <- SUBm(ACC, ASL(ACC2, 1, 0, NULL, 56)); }
            { if (OVF) { L_F <- 1; };
              SET_ALL(OVF, ACC==0, ACC[55]==1, UNORM(ACC), SIGNED(ACC)) }
            { Cycle = 2 + DBM; Size = 1 + DBM ; Stall = 0; }
            { Latency=1; Usage=1; }
    Main_SUBR ACC, ACC2         { Main.OP = 0x06 | (ACC2 << 3); }
            { ACC2 <- SUBm(ACC, ASR(ACC2, 1, 0, NULL, 56)); }
            { if (OVF) { L_F <- 1; };
              SET_ALL(OVF, ACC==0, ACC[55]==1, UNORM(ACC), SIGNED(ACC)) }
            { Cycle = 2 + DBM; Size = 1 + DBM; Stall = 0; }
            { Latency=1; Usage=1; }
// ***** Find some way to describe exception precessing
    Main_SWI                    { Main.OP = 0x06; DBM.OP = 0x00;
                                  DBM.MODE = 0x00; }
            {}
            {}
            { Cycle = 8; Size = 1; Stall = 0; }
            { Latency=1; Usage=1; }
    Main_T C_CODE, REGS, ACC    { Main.OP = 0x00 | (REGS << 4) | (ACC << 3);
                                  DBM.OP = 0x02; DBM.MODE = C_CODE << 4; }
            { if (C_CODE) { ACC <- REGS; }; }
            {}
            { Cycle = 2; Size = 1; Stall = 0; }
            { Latency=1; Usage=1; }
    Main_T C_CODE, REGS, ACC, SONE, DTWO
                                { Main.OP = 0x00 | (REGS << 4) |
                                  (ACC << 3) | DTWO; DBM.OP = 0x03;
                                  DBM.MODE = (C_CODE << 4) | SONE; }
            { if (C_CODE) { ACC <- REGS; DTWO <- SONE; }; }
            {}
            { Cycle = 2; Size = 1; Stall = 0; }
            { Latency=1; Usage=1; }
    Main_TFR REGS, ACC          { Main.OP = 0x01 | (ACC << 3) | (REGS << 4); }
            { ACC <- REGS; }
            {}
            { Cycle = 2 + DBM; Size = 1 + DBM; Stall = 0; }
            { Latency=1; Usage=1; }
    Main_TST ACC                { Main.OP = 0x03 | (ACC << 3); }
            { NULL <- SUBm(ACC, 0); }
            { SET_ALL(FALSE, ACC==0, ACC[55]==1, UNORM(ACC), SIGNED(ACC)) }
            { Cycle = 2 + DBM; Size = 1 + DBM; Stall = 0; }
            { Latency=1; Usage=1; }
    Main_WAIT                   { Main.OP = 0x86; DBM.OP = 0x00;
                                  DBM.MODE = 0x00; }
            { HALT(); }
            {}
// NOTE: The cost is just a dummy given what WAIT does
            { Cycle = 2; Size = 1; Stall = 0; }
            { Latency=1; Usage=1; }
```

```
            { Cycle = 4; Size = 1; Stall = 0; }
            { Latency=1; Usage=1; }
// ***** reset IPR and peripherals
    Main_RESET                  { Main.OP = 0x84; DBM.MODE = 0; DBM.OP = 0; }
            {}
            {}
            { Cycle = 4; Size = 1; Stall = 0; }
            { Latency=1; Usage=1; }
    Main_RND ACC                { Main.OP = 0x11 | (ACC << 3); }
            { ACC <- RNDm(ACC); }
            { if (OVF) { L_F <- 1; };
              SET_ALL(OVF, ACC==0, ACC[55]==1, UNORM(ACC), SIGNED(ACC)) }
            { Cycle = 2 + DBM; Size = 1 + DBM; Stall = 0; }
            { Latency=1; Usage=1; }
    Main_ROL ACCR                { Main.OP = 0x37 | (ACCR << 3); }
            { ACCR[1] <- ROLm(ACCR[1]); }
            { SET_ALL(FALSE, ACCR[1] == 0, ACCR[2][7] == 1, FALSE, FALSE)}
            { Cycle = 2 + DBM; Size = 1 + DBM; Stall = 0; }
            { Latency=1; Usage=1; }
    Main_ROR ACCR                { Main.OP = 0x27 | (ACCR << 3); }
            { ACCR[1] <- RORm(ACCR[1]); }
            { SET_ALL(FALSE, ACCR[1] == 0, ACCR[2][7] == 1, FALSE, FALSE)}
            { Cycle = 2 + DBM; Size = 1 + DBM; Stall = 0; }
            { Latency=1; Usage=1; }
    Main_RTI                    { Main.OP = 0x04; DBM.OP = 0x00;
                                  DBM.MODE = 0x00; }
            { PCS <- SSHS; SRS <- SSLS; SPS <- SPS - 1; }
            {}
            { Cycle = 6; Size = 1; Stall = 0; }
            { Latency=1; Usage=1; }
    Main_RTS                    { Main.OP = 0x0C; DBM.OP = 0x00;
                                  DBM.MODE = 0x00; }
            { PCS <- SSHS; SPS <- SPS - 1; }
            {}
            { Cycle = 6; Size = 1; Stall = 0; }
            { Latency=1; Usage=1; }
    Main_SBC XYSRC, ACC         { Main.OP = 0x25 | (ACC << 3) | (XYSRC << 4); }
            { ACC <- SUBCm(ACC, XYSRC, CCRS[0], CCRS[0]); }
            { if (OVF) { L_F <- 1; };
              SET_ALL(OVF, ACC==0, ACC[55]==1, UNORM(ACC), SIGNED(ACC)) }
            { Cycle = 2 + DBM; Size = 1 + DBM; Stall = 0; }
            { Latency=1; Usage=1; }
    Main_STOP                   { Main.OP = 0x87; DBM.OP = 0x00;
                                  DBM.MODE = 0x00; }
            { HALT(); }
            {}
// NOTE: the cycle Cycle is just a dummy given what STOP does
            { Cycle = 2; Size = 1; Stall = 0; }
            { Latency=1; Usage=1; }
    Main_SUB ALUSRC, ACC        { Main.OP = 0x04 | (ACC << 3) |
                                  (ALUSRC << 4); }
            { ACC <- SUBCm(ACC, ALUSRC, 0, CCRS[0]); }
            { if (OVF) { L_F <- 1; };
              SET_ALL(OVF, ACC==0, ACC[55]==1, UNORM(ACC), SIGNED(ACC)) }
```

```
Main_NORM R_R, ACC          { Main.OP = 0x15 | (ACC << 3);
                              DBM.MODE = 0xD8 | R_R; DBM.OP = 0x01; }
        { if (((~CCRS[5]) & CCRS[4] & (~CCRS[2])) == 1)
             { ACC <- ASLm(ACC); AGU_R[R_R] <- AGU_R[R_R] - 1; } else
             { if (CCRS[5] == 1)
                  { ACC <- ASRm(ACC); AGU_R[R_R] <- AGU_R[R_R] + 1; }; };}
        { if (OVF) { L_F <- 1; };
          SET_ALL(CHANGED(ACC[55]), ACC==0, ACC[55]==1,
                  UNORM(ACC), SIGNED(ACC)) }
        { Cycle = 2; Size = 1; Stall = 0; }
        { Latency=1; Usage=1; }
Main_NOT ACCR                { Main.OP = 0x17 | (ACCR << 3); }
        { ACCR[1] <- NOTm(ACCR[1]); }
        { SET_ALL(FALSE, ACCR[1] == 0, ACCR[2][7] == 1, FALSE, FALSE)}
        { Cycle = 2 + DBM; Size = 1 + DBM; Stall = 0; }
        { Latency=1; Usage=1; }
Main_OR SSR, ACCR            { Main.OP = 0x42 | (ACCR << 3) |
                               (SSR << 4); }
        { ACCR[1] <- ORm(ACCR[1], SSR); }
        { SET_ALL(FALSE, ACCR[1] == 0, ACCR[2][7] == 1, FALSE, FALSE)}
        { Cycle = 2 + DBM; Size = 1 + DBM; Stall = 0; }
        { Latency=1; Usage=1; }
Main_ORI SIMMD, PCREGS       { Main.OP = 0xF8 | PCREGS; DBM.OP = 0;
                               DBM.MODE = SIMMD & 0xFF; }
        { PCREGS <- SORm(PCREGS, SIMMD); }
        {}
        { Cycle = 2; Size = 1; Stall = 0; }
        { Latency=1; Usage=1; }
Main_REP MEMS, ADRM          { Main.OP = 0x20 | (MEMS << 6); DBM.OP = 0x06;
                               DBM.MODE = 0x40 | ADRM; }
        { int 16 TMP; for (TMP <- MEMS[ADRM]; TMP  != 1; TMP<-TMP- 1;) {
                            EVAL(PCS + 1); }; }
        { PCS <- PCS + 2; if (LIMIT) { L_F <- 1; }; ADRM; }
        { Cycle = 4 + ADRM; Size = 1; Stall = 0; }
        { Latency=1; Usage=1; }
Main_REP MEMS, SIMMA         { Main.OP = 0x20 | (MEMS << 6); DBM.OP = 0x06;
                               DBM.MODE = 0x00 | (SIMMA & 0x3F); }
        { int 16 TMP; for (TMP <- MEMS[SIMMA]; TMP  != 1; TMP<-TMP- 1;) {
                            EVAL(PCS + 1); }; }
        { PCS <- PCS + 2; if (LIMIT) { L_F <- 1; }; }
        { Cycle = 4; Size = 1; Stall = 0; }
        { Latency=1; Usage=1; }
Main_REP IMM                 { Main.OP = 0xA0 | ((IMM >> 8) & 0x0F);
                               DBM.OP = 0x06; DBM.MODE = IMM & 0xFF; }
        { int 16 TMP; for (TMP <- IMM; TMP  != 1; TMP<-TMP - 1;) {
                            EVAL(PCS + 1); }; }
        { PCS <- PCS + 2; if (LIMIT) { L_F <- 1; }; }
        { Cycle = 4; Size = 1; Stall = 0; }
        { Latency=1; Usage=1; }
Main_REP DREGS               { Main.OP = 0x20; DBM.OP = 0x06;
                               DBM.MODE = 0xC0 | DREGS; }
        { int 16 TMP; for (TMP <- DREGS; TMP  != 1; TMP<-TMP - 1;) {
                            EVAL(PCS + 1); }; }
        { PCS <- PCS + 2; if (LIMIT) { L_F <- 1; }; }
```

```
                              DBM.MODE = 0x40 | EA; }
        { IM[EA] <- PORTS[PPA]; }
        { EA; }
        { Cycle = 6 + EA; Size = 1 + EA; Stall = 0; }
        { Latency=1; Usage=1; }
Main_MOVEP DREGS, PORTS, PPA
                            { Main.OP = 0x00 | (PPA & 0x3F);
                              DBM.OP = 0x08 | PORTS;
                              DBM.MODE = 0xC0 | DREGS; }
        { PORTS[PPA] <- DREGS; }
        {}
        { Cycle = 6; Size = 1; Stall = 0; }
        { Latency=1; Usage=1; }
Main_MOVEP PORTS, PPA, DREGS
                            { Main.OP = 0x00 | (PPA & 0x3F);
                              DBM.OP = 0x08 | PORTS;
                              DBM.MODE = 0x40 | DREGS; }
        { DREGS <- PORTS[PPA]; }
        {}
        { Cycle = 6; Size = 1; Stall = 0; }
        { Latency=1; Usage=1; }
Main_MPY MREG, ACC     { Main.OP = 0x80 | (ACC << 3) | (MREG << 4); }
        { ACC <- ADDm(0, MREG); }
        { SET_ALL(FALSE, ACC==0, ACC[55]==1, UNORM(ACC), SIGNED(ACC)) }
        { Cycle = 2 + DBM; Size = 1 + DBM; Stall = 0; }
        { Latency=1; Usage=1; }
Main_MPY SIGN, MREG, ACC     { Main.OP = 0x80 | (ACC << 3) |
                               (SIGN << 2) | (MREG << 4); }
        { ACC <- SUBm(0, MREG); }
        { SET_ALL(FALSE, ACC==0, ACC[55]==1, UNORM(ACC), SIGNED(ACC)) }
        { Cycle = 2 + DBM; Size = 1 + DBM; Stall = 0; }
        { Latency=1; Usage=1; }
Main_MPYR MREG, ACC    { Main.OP = 0x81 | (ACC << 3) | (MREG << 4); }
        { ACC <- RNDm(ADDm(0, MREG)); }
        { SET_ALL(FALSE, ACC==0, ACC[55]==1, UNORM(ACC), SIGNED(ACC)) }
        { Cycle = 2 + DBM; Size = 1 + DBM; Stall = 0; }
        { Latency=1; Usage=1; }
Main_MPYR SIGN, MREG, ACC    { Main.OP = 0x81 | (ACC << 3) |
                               (SIGN << 2) | (MREG << 4); }
        { ACC <- RNDm(SUBm(0, MREG)); }
        { SET_ALL(FALSE, ACC==0, ACC[55]==1, UNORM(ACC), SIGNED(ACC)) }
        { Cycle = 2 + DBM; Size = 1 + DBM; Stall = 0; }
        { Latency=1; Usage=1; }
Main_NEG ACC               { Main.OP = 0x36 | (ACC << 3); }
        { ACC <- SUBm(0, ACC); }
        { if (OVF) { L_F <- 1; };
          SET_ALL(OVF, ACC==0, ACC[55]==1, UNORM(ACC), SIGNED(ACC)) }
        { Cycle = 2 + DBM; Size = 1 + DBM; Stall = 0; }
        { Latency=1; Usage=1; }
Main_NOP                   { Main.OP = 0x00; }
        { NOP(); }
        {}
        { Cycle = 2 + DBM; Size = 1 + DBM; Stall = 0; }
        { Latency=1; Usage=1; }
```

```
Main_MOVEM PMEM, EA, DREGS   { Main.OP = 0x80 | DREGS; DBM.OP = 0x07;
                               DBM.MODE = 0x40 | EA; }
        { DREGS <- IM[EA]; }
        { EA; }
        { Cycle = 6 + EA; Size = 1 + EA; Stall = 0; }
        { Latency=1; Usage=1; }
Main_MOVEM DREGS, PMEM, SIMMA  { Main.OP = 0x00 | DREGS; DBM.OP = 0x07;
                               DBM.MODE = 0xC0 | (SIMMA & 0xFF); }
        { IM[SIMMA] <- DREGS; }
        {}
        { Cycle = 6; Size = 1; Stall = 0; }
        { Latency=1; Usage=1; }
Main_MOVEM PMEM, SIMMA, DREGS  { Main.OP = 0x00 | DREGS; DBM.OP = 0x07;
                               DBM.MODE = 0x40 | (SIMMA & 0xFF); }
        { DREGS <- IM[SIMMA]; }
        {}
        { Cycle = 6; Size = 1; Stall = 0; }
        { Latency=1; Usage=1; }
Main_MOVEP MEMS, EA, PORTS, PPA
                             { Main.OP = 0x80 | (PPA & 0x3F) | (MEMS << 6);
                               DBM.OP = 0x08 | PORTS;
                               DBM.MODE = 0xC0 | EA; }
        { PORTS[PPA] <- MEMS[EA]; }
        { EA; }
        { Cycle = 6 + EA; Size = 1 + EA; Stall = 0; }
        { Latency=1; Usage=1; }
Main_MOVEP PORTS, PPA, MEMS, EA
                             { Main.OP = 0x80 | (PPA & 0x3F) | (MEMS << 6);
                               DBM.OP = 0x08 | PORTS;
                               DBM.MODE = 0x40 | EA; }
        { MEMS[EA] <- PORTS[PPA]; }
        { EA; }
        { Cycle = 6 + EA; Size = 1 + EA; Stall = 0; }
        { Latency=1; Usage=1; }
Main_MOVEP LIMMD, PORTS, PPA
                             { Main.OP = 0x80 | (PPA & 0x3F);
                               DBM.OP = 0x08 | PORTS;
                               DBM.MODE = 0xF4;
                               Additional(0, Split.DATA = LIMMD;); }
        { PORTS[PPA] <- LIMMD; }
        {}
        { Cycle = 6; Size = 2; Stall = 0; }
        { Latency=1; Usage=1; }
Main_MOVEP PMEM, EA, PORTS, PPA
                             { Main.OP = 0x40 | (PPA & 0x3F);
                               DBM.OP = 0x08 | PORTS;
                               DBM.MODE = 0xC0 | EA; }
        { PORTS[PPA] <- IM[EA]; }
        { EA; }
        { Cycle = 6; Size = 1 + EA; Stall = 0; }
        { Latency=1; Usage=1; }
Main_MOVEP PORTS, PPA, PMEM, EA
                             { Main.OP = 0x40 | (PPA & 0x3F);
                               DBM.OP = 0x08 | PORTS;
```

```
        { if (LIMIT) { L_F <- 1; }; }
        { Cycle = 2 + EA; Size = 1 + EA; Stall = 0; }
        { Latency=1; Usage=1; }
Main_MOVEC CREG, MEMS, EA    { Main.OP = 0x20 | CREG | (MEMS << 6);
                               DBM.OP = 0x05; DBM.MODE = 0x40 | EA; }
        { MEMS[EA] <- CREG; }
        { EA; }
        { Cycle = 2 + EA; Size = 1 + EA; Stall = 0; }
        { Latency=1; Usage=1; }
Main_MOVEC LIMMD, CREG      { Main.OP = 0x20 | CREG;
                              DBM.OP = 0x05; DBM.MODE = 0xF4;
                              Additional(0, Split.DATA = LIMMD;); }
        { CREG <- LIMMD; }
        {}
        { Cycle = 3; Size = 2; Stall = 0; }
        { Latency=1; Usage=1; }
Main_MOVEC MEMS, SIMMA, CREG   { Main.OP = 0x20 | CREG | (MEMS << 6);
                              DBM.OP = 0x05;
                              DBM.MODE = 0x80 | (SIMMA & 0xFF); }
        { CREG <- MEMS[SIMMA]; }
        {}
        { Cycle = 2; Size = 1; Stall = 0; }
        { Latency=1; Usage=1; }
Main_MOVEC CREG, MEMS, SIMMA   { Main.OP = 0x20 | CREG | (MEMS << 6);
                              DBM.OP = 0x05;
                              DBM.MODE = 0x00 | (SIMMA & 0xFF); }
        { MEMS[SIMMA] <- CREG; }
        {}
        { Cycle = 2; Size = 1; Stall = 0; }
        { Latency=1; Usage=1; }
Main_MOVEC CREG, DREGS      { Main.OP = 0xA0 | CREG; DBM.OP = 0x04;
                              DBM.MODE = 0x40 | DREGS; }
        { DREGS <- CREG; }
        {}
        { Cycle = 2; Size = 1; Stall = 0; }
        { Latency=1; Usage=1; }
Main_MOVEC DREGS, CREG      { Main.OP = 0xA0 | CREG; DBM.OP = 0x04;
                              DBM.MODE = 0xC0 | DREGS; }
        { CREG <- DREGS; }
        {}
        { Cycle = 2; Size = 1; Stall = 0; }
        { Latency=1; Usage=1; }
Main_MOVEC SIMMD, CREG        { Main.OP = 0xA0 | CREG; DBM.OP = 0x05;
                              DBM.MODE = SIMMD & 0xFF; }
        { CREG <- SIMMD; }
        {}
        { Cycle = 2; Size = 1; Stall = 0; }
        { Latency=1; Usage=1; }
Main_MOVEM DREGS, PMEM, EA   { Main.OP = 0x80 | DREGS; DBM.OP = 0x07;
                              DBM.MODE = 0xC0 | EA; }
        { IM[EA] <- DREGS; }
        { EA; }
        { Cycle = 6 + EA; Size = 1 + EA; Stall = 0; }
        { Latency=1; Usage=1; }
```

```
        { Cycle = 8; Size = 2; Stall = 0; }
        { Latency=1; Usage=1; }
Main_JSSET IMM, DREGS, ADDRESS
                            { Main.OP = 0x20 | (IMM & 0x1F); DBM.OP = 0x0B;
                              DBM.MODE = 0xC0 | DREGS;
                              Additional(0, Split.ADDR = ADDRESS;); }
        { if (DREGS[IMM] == 1)
            { SPS <- SPS+1; SSHS <- PCS; SSLS <- SRS; PCS <- ADDRESS; }; }
        {}
        { Cycle = 8; Size = 2; Stall = 0; }
        { Latency=1; Usage=1; }
Main_LSL ACCR                { Main.OP = 0x33 | (ACCR << 3); }
        { ACCR[1] <- LSLm(ACCR[1]); }
        { SET_ALL(FALSE, ACCR[1] == 0, ACCR[1][23] == 1, FALSE, FALSE) }
        { Cycle = 2 + DBM; Size = 1 + DBM; Stall = 0; }
        { Latency=1; Usage=1; }
Main_LSR ACCR                { Main.OP = 0x23 | (ACCR << 3); }
        { ACCR[1] <- LSRm(ACCR[1]); }
        { SET_ALL(FALSE, ACCR[1] == 0, FALSE, FALSE, FALSE) }
        { Cycle = 2 + DBM; Size = 1 + DBM; Stall = 0; }
        { Latency=1; Usage=1; }
Main_LUA SEA, INDEX         { Main.OP = 0x10 | INDEX; DBM.OP = 0x04;
                              DBM.MODE = 0x40 | SEA; }
        { INDEX <- SEA; }
        {}
        { Cycle = 4; Size = 1; Stall = 0; }
        { Latency=1; Usage=1; }
Main_MAC MREG, ACC     { Main.OP = 0x82 | (ACC << 3) | (MREG << 4); }
        { ACC <- ADDm(ACC, MREG); }
        { if (OVF) { L_F <- 1; };
          SET_ALL(OVF, ACC==0, ACC[55]==1, UNORM(ACC), SIGNED(ACC)) }
        { Cycle = 2 + DBM; Size = 1 + DBM; Stall = 0; }
        { Latency=1; Usage=1; }
Main_MAC SIGN, MREG, ACC     { Main.OP = 0x82 | (ACC << 3) |
                               (SIGN << 2) | (MREG << 4); }
        { ACC <- SUBm(ACC, MREG); }
        { if (OVF) { L_F <- 1; };
          SET_ALL(OVF, ACC==0, ACC[55]==1, UNORM(ACC), SIGNED(ACC)) }
        { Cycle = 2 + DBM; Size = 1 + DBM; Stall = 0; }
        { Latency=1; Usage=1; }
Main_MACR MREG, ACC    { Main.OP = 0x83 | (ACC << 3) | (MREG << 4); }
        { ACC <- RNDm(ADDm(ACC, MREG)); }
        {}
        { Cycle = 2 + DBM; Size = 1 + DBM; Stall = 0; }
        { Latency=1; Usage=1; }
Main_MACR SIGN, MREG, ACC    { Main.OP = 0x83 | (ACC << 3) |
                               (SIGN << 2) | (MREG << 4); }
        { ACC <- RNDm(SUBm(ACC, MREG)); }
        {}
        { Cycle = 2 + DBM; Size = 1 + DBM; Stall = 0; }
        { Latency=1; Usage=1; }
Main_MOVEC MEMS, EA, CREG    { Main.OP = 0x20 | CREG | (MEMS << 6);
                               DBM.OP = 0x05; DBM.MODE = 0xC0 | EA; }
        { CREG <- MEMS[EA]; }
```

```
                            DBM.MODE = 0x80 | PPA;
                            Additional(0, Split.ADDR = ADDRESS;); }
        { if (PORTS[PPA][IMM] == 1)  { PCS <- ADDRESS; }; }
        {}
        { Cycle = 8; Size = 2; Stall = 0; }
        { Latency=1; Usage=1; }
Main_JSET IMM, DREGS, ADDRESS
                           { Main.OP = 0x20 | (IMM & 0x1F); DBM.OP = 0x0A;
                            DBM.MODE = 0xC0 | DREGS;
                            Additional(0, Split.ADDR = ADDRESS;); }
        { if (DREGS[IMM] == 1)  { PCS <- ADDRESS; }; }
        {}
        { Cycle = 8; Size = 2; Stall = 0; }
        { Latency=1; Usage=1; }
Main_JSRI ADDRESS            { Main.OP = ADDRESS & 0xFF; DBM.OP = 0xD0;
                            DBM.MODE = (ADDRESS >> 8) & 0xF; }
        { SPS <- SPS + 1; SSHS <- PCS; SSLS <- SRS; PCS <- ADDRESS; }
        {}
        { Cycle = 4; Size = 1; Stall = 0; }
        { Latency=1; Usage=1; }
Main_JSR EA                 { Main.OP = 0x80; DBM.OP = 0xC0;
                            DBM.MODE = 0x0C | EA; }
        { SPS <- SPS + 1; SSHS <- PCS; SSLS <- SRS; PCS <- EA; }
        { EA; }
        { Cycle = 4 + EA; Size = 1 + EA; Stall = 0; }
        { Latency=1; Usage=1; }
Main_JSSET IMM, MEMS, ADRM, ADDRESS
                           { Main.OP = 0xA0 | (IMM & 0x1F) |
                            (MEMS << 6); DBM.OP = 0x0B;
                            DBM.MODE = 0x40 | ADRM;
                            Additional(0, Split.ADDR = ADDRESS;); }
        { if (MEMS[ADRM][IMM] == 1)
            { SPS <- SPS+1; SSHS <- PCS; SSLS <- SRS; PCS <- ADDRESS; }; }
        { ADRM; }
        { Cycle = 6 + ADRM; Size = 2; Stall = 0; }
        { Latency=1; Usage=1; }
Main_JSSET IMM, MEMS, SIMMA, ADDRESS
                           { Main.OP = 0xA0 | (IMM & 0x1F) |
                            (MEMS << 6); DBM.OP = 0x0B;
                            DBM.MODE = 0x00 | (SIMMA & 0x3F);
                            Additional(0, Split.ADDR = ADDRESS;); }
        { if (MEMS[SIMMA][IMM] == 1)
            { SPS <- SPS+1; SSHS <- PCS; SSLS <- SRS; PCS <- ADDRESS; }; }
        {}
        { Cycle = 8; Size = 2; Stall = 0; }
        { Latency=1; Usage=1; }
Main_JSSET IMM, PORTS, PPA, ADDRESS
                           { Main.OP = 0xA0 | (IMM & 0x1F) |
                            (PORTS << 6); DBM.OP = 0x0B;
                            DBM.MODE = 0x80 | PPA;
                            Additional(0, Split.ADDR = ADDRESS;); }
        { if (PORTS[PPA][IMM] == 1)
            { SPS <- SPS+1; SSHS <- PCS; SSLS <- SRS; PCS <- ADDRESS; }; }
        {}
```

```
        { if (MEMS[ADRM][IMM] == 0)
            { SPS <- SPS+1; SSHS <- PCS; SSLS <- SRS; PCS <- ADDRESS; }; }
        { ADRM; }
        { Cycle = 6 + ADRM; Size = 2; Stall = 0; }
        { Latency=1; Usage=1; }
Main_JSCLR IMM, MEMS, SIMMA, ADDRESS
                          { Main.OP = 0x80 | (IMM & 0x1F) |
                            (MEMS << 6); DBM.OP = 0x0B;
                            DBM.MODE = 0x00 | (SIMMA & 0x3F);
                            Additional(0, Split.ADDR = ADDRESS;); }
        { if (MEMS[SIMMA][IMM] == 0)
            { SPS <- SPS+1; SSHS <- PCS; SSLS <- SRS; PCS <- ADDRESS; }; }
        {}
        { Cycle = 8; Size = 2; Stall = 0; }
        { Latency=1; Usage=1; }
Main_JSCLR IMM, PORTS, PPA, ADDRESS
                          { Main.OP = 0x80 | (IMM & 0x1F) |
                            (PORTS << 6); DBM.OP = 0x0B;
                            DBM.MODE = 0x80 | PPA;
                            Additional(0, Split.ADDR = ADDRESS;); }
        { if (PORTS[PPA][IMM] == 0)
            { SPS <- SPS+1; SSHS <- PCS; SSLS <- SRS; PCS <- ADDRESS; }; }
        {}
        { Cycle = 8; Size = 2; Stall = 0; }
        { Latency=1; Usage=1; }
Main_JSCLR IMM, DREGS, ADDRESS
                          { Main.OP = 0x00 | (IMM & 0x1F); DBM.OP = 0x0B;
                            DBM.MODE = 0xC0 | DREGS;
                            Additional(0, Split.ADDR = ADDRESS;); }
        { if (DREGS[IMM] == 0)
            { SPS <- SPS+1; SSHS <- PCS; SSLS <- SRS; PCS <- ADDRESS; }; }
        {}
        { Cycle = 8; Size = 2; Stall = 0; }
        { Latency=1; Usage=1; }
Main_JSET IMM, MEMS, ADRM, ADDRESS
                          { Main.OP = 0xA0 | (IMM & 0x1F) |
                            (MEMS << 6); DBM.OP = 0x0A;
                            DBM.MODE = 0x40 | ADRM; }
        { if (MEMS[ADRM][IMM] == 1)  { PCS <- ADDRESS; }; }
        { ADRM; }
        { Cycle = 6 + ADRM; Size = 2; Stall = 0; }
        { Latency=1; Usage=1; }
Main_JSET IMM, MEMS, SIMMA, ADDRESS
                          { Main.OP = 0xA0 | (IMM & 0x1F) |
                            (MEMS << 6); DBM.OP = 0x0A;
                            DBM.MODE = 0x00 | (SIMMA & 0x3F);
                            Additional(0, Split.ADDR = ADDRESS;); }
        { if (MEMS[SIMMA][IMM] == 1)  { PCS <- ADDRESS; }; }
        {}
        { Cycle = 8; Size = 2; Stall = 0; }
        { Latency=1; Usage=1; }
Main_JSET IMM, PORTS, PPA, ADDRESS
                          { Main.OP = 0xA0 | (IMM & 0x1F) |
                            (PORTS << 6); DBM.OP = 0x0A;
```

```
                            DBM.MODE = 0x00 | (SIMMA & 0x3F);
                            Additional(0, Split.ADDR = ADDRESS;); }
        { if (MEMS[SIMMA][IMM]==0) { PCS<-ADDRESS; }; }
        {}
        { Cycle = 8; Size = 2; Stall = 0; }
        { Latency=1; Usage=1; }
Main_JCLR IMM, PORTS, PPA, ADDRESS
                           { Main.OP = 0x80 | (IMM & 0x1F) |
                            (PORTS << 6); DBM.OP = 0x0A;
                            DBM.MODE = 0x80 | PPA;
                            Additional(0, Split.ADDR = ADDRESS;); }
        { if (PORTS[PPA][IMM]==0) { PCS<-ADDRESS; }; }
        {}
        { Cycle = 8; Size = 2; Stall = 0; }
        { Latency=1; Usage=1; }
Main_JCLR IMM, DREGS, ADDRESS
                           { Main.OP = 0x00 | (IMM & 0x1F); DBM.OP = 0x0A;
                            DBM.MODE = 0xC0 | DREGS;
                            Additional(0, Split.ADDR = ADDRESS;); }
        { if (DREGS[IMM]==0) { PCS<-ADDRESS; }; }
        {}
        { Cycle = 8; Size = 2; Stall = 0; }
        { Latency=1; Usage=1; }
Main_JMPI ADDRESS           { Main.OP = ADDRESS & 0xFF; DBM.OP = 0x0C;
                            DBM.MODE = (ADDRESS >> 8) & 0xF; }
        { PCS <- ADDRESS; }
        {}
        { Cycle = 4; Size = 1; Stall = 0; }
        { Latency=1; Usage=1; }
Main_JMP EA                 { Main.OP = 0x80; DBM.OP = 0x0A;
                            DBM.MODE = 0xC0 | EA; }
        { PCS <- EA; }
        { EA; }
        { Cycle = 4 + EA; Size = 1 + EA; Stall = 0; }
        { Latency=1; Usage=1; }
Main_JSI C_CODE, ADDRESS    { Main.OP = ADDRESS & 0xFF; DBM.OP = 0x0F;
                            DBM.MODE = C_CODE | ((ADDRESS>>8) & 0x0F); }
        { if (C_CODE)
            { SPS <- SPS+1; SSHS <- PCS; SSLS <- SRS; PCS <- ADDRESS; }; }
        {}
        { Cycle = 4; Size = 1; Stall = 0; }
        { Latency=1; Usage=1; }
Main_JS C_CODE, EA          { Main.OP = 0xA0 | C_CODE ; DBM.OP = 0x0B;
                            DBM.MODE = 0xC0 | EA; }
        { if (C_CODE)
            { SPS <- SPS+1; SSHS <- PCS; SSLS <- SRS; PCS <- EA; }; }
        { EA; }
        { Cycle = 4 + EA; Size = 1 + EA; Stall = 0; }
        { Latency=1; Usage=1; }
Main_JSCLR IMM, MEMS, ADRM, ADDRESS
                           { Main.OP = 0x80 | (IMM & 0x1F) |
                            (MEMS << 6); DBM.OP = 0x0B;
                            DBM.MODE = 0x40 | ADRM;
                            Additional(0, Split.ADDR = ADDRESS;); }
```

```
              SPS <- SPS + 1; SSHS <- PCS; SSLS <- SRS; LAS <- ADDRESS; }
            { LF_F <- 1; }
            { Cycle = 8; Size = 2; Stall = 0; }
            { Latency=1; Usage=1; }
    Main_DO DREGS, ADDRESS      { Main.OP = 0x00;
                                  DBM.MODE = 0xC0 | DREGS; DBM.OP = 0x06; }
            { SPS <- SPS + 1; SSHS <- LAS; SSLS <- LCS; LCS <- DREGS;
              SPS <- SPS + 1; SSHS <- PCS; SSLS <- SRS; LAS <- ADDRESS; }
            { LF_F <- 1; }
            { Cycle = 8; Size = 2; Stall = 0; }
            { Latency=1; Usage=1; }
    Main_ENDDO                  { Main.OP = 0x8C; DBM.MODE = 0x00;
                                  DBM.OP = 0x00; }
            { LF_F <- SSLS[15]; SPS <- SPS - 1;
              LAS <- SSHS; LCS <- SSLS; SPS <- SPS - 1; }
            {}
            { Cycle = 2; Size = 1; Stall = 0; }
            { Latency=1; Usage=1; }
    Main_EOR SSR, ACCR           { Main.OP = 0x43 | (ACCR << 3) | (SSR << 4); }
            { ACCR[1] <- XORm(ACCR[1], SSR); }
            { SET_ALL(FALSE, ACCR[1] == 0, ACCR[1][23] == 1, FALSE, FALSE) }
            { Cycle = 2 + DBM; Size = 1 + DBM; Stall = 0; }
            { Latency=1; Usage=1; }
// ***** Also figure out the RTL for this
    Main_ILLEGAL                { Main.OP = 0x05; DBM.MODE = 0x00;
                                  DBM.OP = 0x00; }
            {}
            {}
            { Cycle = 8; Size = 1; Stall = 0; }
            { Latency=1; Usage=1; }
    Main_JI C_CODE, ADDRESS     { Main.OP = ADDRESS & 0xFF; DBM.OP = 0x0E;
                                  DBM.MODE = C_CODE | ((ADDRESS>>8) & 0x0F); }
            { if (C_CODE) { PCS <- ADDRESS;}; }
            {}
            { Cycle = 4; Size = 1; Stall = 0; }
            { Latency=1; Usage=1; }
    Main_J C_CODE, EA           { Main.OP = 0xA0 | C_CODE; DBM.OP = 0x0A;
                                  DBM.MODE = 0xC0 | EA; }
            { if (C_CODE) { PCS <- EA;}; }
            { EA; }
            { Cycle = 4 + EA; Size = 1 + EA; Stall = 0; }
            { Latency=1; Usage=1; }
    Main_JCLR IMM, MEMS, ADRM, ADDRESS
                                { Main.OP = 0x80 | (IMM & 0x1F) |
                                  (MEMS << 6); DBM.OP = 0x0A;
                                  DBM.MODE = 0x40 | ADRM;
                                  Additional(0, Split.ADDR = ADDRESS;); }
            { if (MEMS[ADRM][IMM]==0) { PCS <- ADDRESS; }; }
            { ADRM; }
            { Cycle = 6 + ADRM; Size = 2; Stall = 0; }
            { Latency=1; Usage=1; }
    Main_JCLR IMM, MEMS, SIMMA, ADDRESS
                                { Main.OP = 0x80 | (IMM & 0x1F) |
                                  (MEMS << 6); DBM.OP = 0x0A;
```

```
    Main_BTST IMM, PORTS, PPA   { Main.OP = 0x20 | (IMM & 0x1F) |
                                  (PORTS << 6); DBM.OP = 0x0B;
                                  DBM.MODE = 0x80 | PPA; }
            { C_F <- PORTS[PPA][IMM]; }
            {}
            { Cycle = 4; Size = 1; Stall = 0; }
            { Latency=1; Usage=1; }
    Main_BTST IMM, DREGS        { Main.OP = 0x20 | (IMM & 0x1F); DBM.OP = 0x0B;
                                  DBM.MODE = 0xC0 | DREGS; }
            { C_F <- DREGS[IMM]; }
            {}
            { Cycle = 4; Size = 1; Stall = 0; }
            { Latency=1; Usage=1; }
    Main_CLR ACC                { Main.OP = 0x13 | (ACC << 3); }
            { ACC <- 0; }
            { CLEAR_ALL(0, 1, 0, 1, 0) }
            { Cycle = 2 + DBM; Size = 1 + DBM; Stall = 0; }
            { Latency=1; Usage=1; }
    Main_CMP REGS, ACC          { Main.OP = 0x05 | (ACC << 3) | (REGS << 4); }
            { NULL <- SUBm(REGS, ACC); }
            {}
            { Cycle = 2 + DBM; Size = 1 + DBM; Stall = 0; }
            { Latency=1; Usage=1; }
    Main_CMPM REGS, ACC         { Main.OP = 0x07 | (ACC << 3) | (REGS << 4); }
            { NULL <- SUBm(ABSm(REGS), ABSm(ACC)); }
            { if (OVF) { L_F <- 1; };
              SET_ALL(OVF, ACC==0, ACC[55]==1, UNORM(ACC), SIGNED(ACC)) }
            { Cycle = 2 + DBM; Size = 1 + DBM; Stall = 0; }
            { Latency=1; Usage=1; }
// ***** Also figure out how to do the RTL
    Main_DIV SSR, ACC           { Main.OP = 0x20 | (ACC << 3) | (SSR << 4);
                                  DBM.MODE = 0x80; DBM.OP = 0x01; }
            {}
            { L_F <- V_F; SET_FLAG(CHANGED(MS(ACC)), V_F);
              SET_FLAG(CLEARED(ACC[55]), C_F);}
            { Cycle = 2; Size = 1; Stall = 0; }
            { Latency=1; Usage=1; }
    Main_DO MEMS, ADRM, ADDRESS { Main.OP = 0x00 | (MEMS << 6);
                                  DBM.MODE = 0x40 | ADRM; DBM.OP = 0x06; }
            { SPS <- SPS + 1; SSHS <- LAS; SSLS <- LCS; LCS <- MEMS[ADRM];
              SPS <- SPS + 1; SSHS <- PCS; SSLS <- SRS; LAS <- ADDRESS; }
            { LF_F <- 1; EA; }
            { Cycle = 6 + ADRM; Size = 2; Stall = 0; }
            { Latency=1; Usage=1; }
    Main_DO MEMS, SIMMA, ADDRESS   { Main.OP = 0x00 | (MEMS << 6);
                                  DBM.MODE = 0x00 | SIMMA; DBM.OP = 0x06; }
            { SPS <- SPS + 1; SSHS <- LAS; SSLS <- LCS; LCS <- MEMS[SIMMA];
              SPS <- SPS + 1; SSHS <- PCS; SSLS <- SRS; LAS <- ADDRESS; }
            { LF_F <- 1; }
            { Cycle = 8; Size = 2; Stall = 0; }
            { Latency=1; Usage=1; }
    Main_DO IMM, ADDRESS        { Main.OP = 0x80 | ((IMM >> 8) & 0xF);
                                  DBM.MODE = IMM & 0xFF; DBM.OP = 0x06; }
            { SPS <- SPS + 1; SSHS <- LAS; SSLS <- LCS; LCS <- IMM;
```

```
Main_BCLR IMM, PORTS, PPA   { Main.OP = 0x00 | (IMM & 0x1F) |
                              (PORTS << 6); DBM.OP = 0x0A;
                              DBM.MODE = 0x80 | PPA; }
        { PORTS[PPA][IMM] <- 0; }
        { SET_FLAG(~PORTS[PPA][IMM], C_F); }
        { Cycle = 6; Size = 1; Stall = 0; }
        { Latency=1; Usage=1; }
Main_BCLR IMM, DREGS        { Main.OP = 0x00 | (IMM & 0x1F); DBM.OP = 0x0A;
                              DBM.MODE = 0xC0 | DREGS; }
        { DREGS[IMM] <- 0; }
        { SET_FLAG(~DREGS[IMM], C_F); }
        { Cycle = 4; Size = 1; Stall = 0; }
        { Latency=1; Usage=1; }
Main_BSET IMM, MEMS, EA      { Main.OP = 0x20 | (IMM & 0x1F) |
                              (MEMS << 6); DBM.OP = 0x0A;
                              DBM.MODE = 0x40 | EA; }
        { MEMS[EA][IMM] <- 1; }
        { SET_FLAG(~MEMS[EA][IMM], C_F); EA; }
        { Cycle = 4 + EA; Size = 1 + EA; Stall = 0; }
        { Latency=1; Usage=1; }
Main_BSET IMM, MEMS, SIMMA  { Main.OP = 0x20 | (IMM & 0x1F) |
                              (MEMS << 6); DBM.OP = 0x0A;
                              DBM.MODE = 0x00 | (SIMMA & 0x3F); }
        { MEMS[SIMMA][IMM] <- 1; }
        { SET_FLAG(~MEMS[SIMMA][IMM], C_F); }
        { Cycle = 4; Size = 1; Stall = 0; }
        { Latency=1; Usage=1; }
Main_BSET IMM, PORTS, PPA   { Main.OP = 0x20 | (IMM & 0x1F) |
                              (PORTS << 6); DBM.OP = 0x0A;
                              DBM.MODE = 0x80 | PPA; }
        { PORTS[PPA][IMM] <- 1; }
        { SET_FLAG(~PORTS[PPA][IMM], C_F); }
        { Cycle = 6; Size = 1; Stall = 0; }
        { Latency=1; Usage=1; }
Main_BSET IMM, DREGS        { Main.OP = 0x20 | (IMM & 0x1F); DBM.OP = 0x0A;
                              DBM.MODE = 0xC0 | DREGS; }
        { DREGS[IMM] <- 1; }
        { SET_FLAG(~DREGS[IMM], C_F); }
        { Cycle = 4; Size = 1; Stall = 0; }
        { Latency=1; Usage=1; }
Main_BTST IMM, MEMS, EA      { Main.OP = 0x20 | (IMM & 0x1F) |
                              (MEMS << 6); DBM.OP = 0x0B;
                              DBM.MODE = 0x40 | EA; }
        { C_F <- MEMS[EA][IMM]; }
        { EA; }
        { Cycle = 4 + EA; Size = 1 + EA; Stall = 0; }
        { Latency=1; Usage=1; }
Main_BTST IMM, MEMS, SIMMA  { Main.OP = 0x20 | (IMM & 0x1F) |
                              (MEMS << 6); DBM.OP = 0x0B;
                              DBM.MODE = 0x00 | (SIMMA & 0x3F); }
        { C_F <- MEMS[SIMMA][IMM]; }
        {}
        { Cycle = 4; Size = 1; Stall = 0; }
        { Latency=1; Usage=1; }
```

```
Main_ASL ACC                { Main.OP = 0x32 | (ACC << 3); }
        { ACC <- ASLm(ACC); }
        { if (OVF) { L_F <- 1; };
          SET_ALL(CHANGE(ACC[55]), ACC==0, ACC[55]==1,
                    UNORM(ACC), SIGNED(ACC)) }
        { Cycle = 2 + DBM; Size = 1 + DBM; Stall = 0; }
        { Latency=1; Usage=1; }
Main_ASR ACC                { Main.OP = 0x22 | (ACC << 3); }
        { ACC <- ASRm(ACC); }
        { SET_ALL(FALSE, ACC==0, ACC[55]==1,
                              UNORM(ACC), SIGNED(ACC)) }
        { Cycle = 2 + DBM; Size = 1 + DBM; Stall = 0; }
        { Latency=1; Usage=1; }
Main_BCHG IMM, MEMS, EA     { Main.OP = 0x00 | (IMM & 0x1F) |
                              (MEMS << 6); DBM.OP = 0x0D;
                              DBM.MODE =  0x40 | EA; }
        { MEMS[EA][IMM] <- ~MEMS[EA][IMM]; }
        { SET_FLAG(~MEMS[EA][IMM], C_F); EA; }
        { Cycle = 4 + EA; Size = 1 + EA; Stall = 0; }
        { Latency=1; Usage=1; }
Main_BCHG IMM, MEMS, SIMMA   { Main.OP = 0x00 | (IMM & 0x1F) |
                              (MEMS << 6); DBM.OP = 0x0D;
                              DBM.MODE = 0x00 | (SIMMA & 0x3F);}
        { MEMS[SIMMA][IMM] <- ~MEMS[SIMMA][IMM]; }
        { SET_FLAG(~MEMS[SIMMA][IMM], C_F); }
        { Cycle = 4; Size = 1; Stall = 0; }
        { Latency=1; Usage=1; }
Main_BCHG IMM, PORTS, PPA    { Main.OP = 0x00 | (IMM & 0x1F) |
                              (PORTS << 6); DBM.OP = 0x0D;
                              DBM.MODE = 0x80 | PPA; }
        { PORTS[PPA][IMM] <- ~PORTS[PPA][IMM]; }
        { SET_FLAG(~PORTS[PPA][IMM], C_F); }
        { Cycle = 6; Size = 1; Stall = 0; }
        { Latency=1; Usage=1; }
Main_BCHG IMM, DREGS        { Main.OP = 0x00 | (IMM & 0x1F); DBM.OP = 0x0D;
                              DBM.MODE = 0xC0 | DREGS; }
        { DREGS[IMM] <- ~DREGS[IMM]; }
        { SET_FLAG(~DREGS[IMM], C_F); }
        { Cycle = 4; Size = 1; Stall = 0; }
        { Latency=1; Usage=1; }
Main_BCLR IMM, MEMS, EA      { Main.OP = 0x00 | (IMM & 0x1F) |
                              (MEMS << 6); DBM.OP = 0x0A;
                              DBM.MODE = 0x40 | EA; }
        { MEMS[EA][IMM] <- 0; }
        { SET_FLAG(~MEMS[EA][IMM], C_F); EA; }
        { Cycle = 4 + EA; Size = 1 + EA; Stall = 0; }
        { Latency=1; Usage=1; }
Main_BCLR IMM, MEMS, SIMMA  { Main.OP = 0x00 | (IMM & 0x1F) |
                              (MEMS << 6); DBM.OP = 0x0A;
                              DBM.MODE = 0x00 | (SIMMA & 0x3F); }
        { MEMS[SIMMA][IMM] <- 0; }
        { SET_FLAG(~MEMS[SIMMA][IMM], C_F);}
        { Cycle = 4; Size = 1; Stall = 0; }
        { Latency=1; Usage=1; }
```

```
// NOTE: The timing described here assumes there is no wait states
//       and all memory is internal.

// ***** Handle the before/after operation issue on side effects

Section Instruction_Set

Field Main:
    Main_NULL                   DEFINE_NULL_OP
    Main_ABS ACC                { Main.OP = 0x26 | (ACC << 3); }
            { ACC <- ABSm(ACC); }
            { if (OVF) { L_F <- 1; };
              SET_ALL(OVF, ACC==0, ACC[55]==1, UNORM(ACC), SIGNED(ACC)) }
            { Cycle = 2 + DBM; Size = 1 + DBM; Stall = 0; }
            { Latency=1; Usage=1; }
    Main_ADC XYSRC, ACC         { Main.OP = 0x21 | (ACC << 3) |
                                    (XYSRC << 4); }
            { ACC <- ADDCm(ACC, EXT(XYSRC,48,56), CCRS[0], CCRS[0]); }
            { if (OVF) { L_F <- 1; };
              SET_ALL(OVF, ACC==0, ACC[55]==1, UNORM(ACC), SIGNED(ACC)) }
            { Cycle = 2 + DBM; Size = 1 + DBM; Stall = 0; }
            { Latency=1; Usage=1; }
    Main_ADD ALUSRC, ACC        { Main.OP = 0x00 | (ACC << 3) |
                                    (ALUSRC << 4); }
            { ACC <- ADDCm(ACC, EXT(ALUSRC,48,56), 0, CCRS[0]); }
            { if (OVF) { L_F <- 1; };
              SET_ALL(OVF, ACC==0, ACC[55]==1, UNORM(ACC), SIGNED(ACC)) }
            { Cycle = 2 + DBM; Size = 1 + DBM; Stall = 0; }
            { Latency=1; Usage=1; }
    Main_ADDL ACC, ACC2          { Main.OP = 0x12 | (ACC2 << 3); }
            { ACC2 <- ADDm(ACC, ASL(ACC2, 1, 0, NULL, 56)); }
            { if (OVF) { L_F <- 1; };
              SET_ALL(OVF, ACC==0, ACC[55]==1, UNORM(ACC), SIGNED(ACC)) }
            { Cycle = 2 + DBM; Size = 1 + DBM; Stall = 0; }
            { Latency=1; Usage=1; }
    Main_ADDR ACC, ACC2         { Main.OP = 0x02 | (ACC2 << 3); }
            { ACC2 <- ADDm(ACC, ASR(ACC2, 1, 0, NULL, 56)); }
            { if (OVF) { L_F <- 1; };
              SET_ALL(OVF, ACC==0, ACC[55]==1, UNORM(ACC), SIGNED(ACC)) }
            { Cycle = 2 + DBM; Size = 1 + DBM; Stall = 0; }
            { Latency=1; Usage=1; }
    Main_AND SSR, ACCR           { Main.OP = 0x46 | (ACCR << 3) |
                                    (SSR << 4); }
            { ACCR[1] <- ANDm(ACCR[1], SSR); }
            {}
            { Cycle = 2 + DBM; Size = 1 + DBM; Stall = 0; }
            { Latency=1; Usage=1; }
    Main_ANDI IMM, PCREGS       { Main.OP = 0xb8 | PCREGS; DBM.MODE =
                                  IMM & 0xff; }
            { PCREGS <- ANDSm(PCREGS, IMM); }
            { }
            { Cycle = 2; Size = 1; Stall = 0; }
            { Latency=1; Usage=1; }
```

```
                '(' R_R ')'         {$$ = 0x00 | (R_R & 0x3); } {AGU_R[R_R]}
                    {} { Cycle = 2; Size = 0; } {};

RENAME(A_D, A_D1);
RENAME(B_D, B_D1);

Split.ADDR      DBM.OP+DBM.MODE+Main.OP;
Split.DATA      DBM.OP+DBM.MODE+Main.OP;

// ---------------------------------------------------------------------------
// The number of registers for each unit and the topology of
// any of the register files.

Section Storage

//                      = depth , width
Instruction Memory IM   = 0x1000 , 0x18
Memory  XMEMS           = 0x1000 , 0x18
Memory  YMEMS           = 0x1000 , 0x18
RegFile AGU_R           = 0x8 , 0x18
RegFile AGU_N           = 0x8 , 0x18
RegFile AGU_M           = 0x8 , 0x18
RegFile X               = 0x2 , 0x18
RegFile Y               = 0x2 , 0x18
RegFile A               = 0x3 , 0x18
RegFile B               = 0x3 , 0x18
MMIO XPort              = 0x40 , 0x18
MMIO YPort              = 0x40 , 0x18
CRegister LAS           = 0x10          // Loop Address
CRegister LCS           = 0x10          // Loop Counter
CRegister MRS           = 0x8
CRegister CCRS          = 0x8
CRegister OMRS          = 0x8
CRegister SPS           = 0x6
ProgramCounter PCS      = 0x10
Stack SSS(SPS)          = 0xf , 0x20

Alias AREG      A[2][0x0 - 0x7],A[1],A[0];          0x38
Alias BREG      B[2][0x0 - 0x7],B[1],B[0];          0x38
Alias XREG      X[1],X[0];                          0x30
Alias YREG      Y[1],Y[0];                          0x30
Alias SSHS      SSS[SPS][0x10 - 0x1f];              0x10
Alias SSLS      SSS[SPS][0x00 - 0x0f];              0x10
Alias SRS       MRS,CCRS;                           0x10
Alias A10       A[1],A[0];                          0x30
Alias B10       B[1],B[0];                          0x30
Alias LMEMS     XMEMS[0 - 0xFFF],YMEMS[0 - 0xFFF];  0x1000,0x30
Alias ABREG     A[1],B[1];                          0x30
Alias BAREG     B[1],A[1];                          0x30
Alias S_MODE    MRS[2 - 3];                         0x02

// ---------------------------------------------------------------------------
// Correspondence between assembly mneumonics, bitfields, and actual
// instructions.
```

```
                  X0_R ',' X0_R1 { $$ = 0x0; } {MULm(X[0], X[0])} {} {} {} |
                  Y0_R ',' Y0_R1 { $$ = 0x1; } {MULm(Y[0], Y[0])} {} {} {} |
                  X1_R ',' X0_R { $$ = 0x2; } {MULm(X[1], X[0])} {} {} {} |
                  Y1_R ',' Y0_R { $$ = 0x3; } {MULm(Y[1], Y[0])} {} {} {} |
                  X0_R ',' Y1_R { $$ = 0x4; } {MULm(X[0], Y[1])} {} {} {} |
                  Y0_R ',' X0_R { $$ = 0x5; } {MULm(Y[0], X[0])} {} {} {} |
                  X1_R ',' Y0_R { $$ = 0x6; } {MULm(X[1], Y[0])} {} {} {} |
                  Y1_R ',' X1_R { $$ = 0x7; } {MULm(Y[1], X[1])} {} {} {} ;
Non_Terminal      CREG:
                  M_R {$$ = M_R; } {AGU_M[M_R]} {AGU_M[M_R]} {} {} |
                  PROGREGS {$$ = 0x18 | PROGREGS; } {PROGREGS}
                              {PROGREGS} {} {} ;
Non_Terminal      MEMS:
                  XMEM {$$ = 0;} {XMEMS} {XMEMS} {} {} |
                  YMEM {$$ = 1;} {YMEMS} {YMEMS} {} {} ;
Non_Terminal      PORTS:
                  XMEM {$$ = 0;} {XPort} {XPort} {} {} |
                  YMEM {$$ = 1;} {YPort} {YPort} {} {} ;
Non_Terminal      SONE: R_R {$$ = R_R; } {AGU_R[R_R]} {AGU_R[R_R]} {} {} ;
Non_Terminal      DTWO: R_R {$$ = R_R; } {AGU_R[R_R]} {AGU_R[R_R]} {} {} ;
Non_Terminal      XA:
                  X_R {$$ = X_R; } {X[X_R]} {X[X_R]} {} {} |
                  A_D {$$ = 2; }  {AREG} {AREG} {} {} |
                  B_D {$$ = 3; }  {BREG} {BREG} {} {} ;
Non_Terminal      YA:
                  Y_R {$$ = Y_R; } {Y[Y_R]} {Y[Y_R]} {} {} |
                  A_D {$$ = 2; }  {AREG} {AREG} {} {} |
                  B_D {$$ = 3; }  {BREG} {BREG} {} {} ;
Non_Terminal      LREG:
                  A10_D {$$ = 0; } {A10}      {A10} {} {} |
                  B10_D {$$ = 1; } {B10}      {B10} {} {} |
                  X_D   {$$ = 2; } {XREG}     {XREG} {} {} |
                  Y_D   {$$ = 3; } {YREG}     {YREG} {} {} |
                  A_D   {$$ = 4; } {AREG}     {AREG} {} {} |
                  B_D   {$$ = 5; } {BREG}     {BREG} {} {} |
                  AB_D  {$$ = 6; } {ABREG}    {ABREG} {} {} |
                  BA_D  {$$ = 7; } {BAREG}    {BAREG} {} {} ;
Non_Terminal      DXEA:
                '(' R_R ')' '+' N_R {$$ = 0x08 | (R_R & 0x7); } {AGU_R[R_R]}
                   { AGU_R[R_R] <- AGU_R[R_R] + AGU_N[N_R]; } {} {} |
                '(' R_R ')' '-'     {$$ = 0x10 | (R_R & 0x7); } {AGU_R[R_R]}
                   { AGU_R[R_R] <- AGU_R[R_R] - 1; }   {} {} |
                '(' R_R ')' '+'     {$$ = 0x18 | (R_R & 0x7); } {AGU_R[R_R]}
                   { AGU_R[R_R] <- AGU_R[R_R] + 1; }
                       { Cycle = 2; Size = 0; } {} |
                '(' R_R ')'         {$$ = 0x00 | (R_R & 0x7); } {AGU_R[R_R]}
                   {} { Cycle = 2; Size = 0; } {} ;
Non_Terminal      DYEA:
                '(' R_R ')' '+' N_R {$$ = 0x04 | (R_R & 0x3); } {AGU_R[R_R]}
                   { AGU_R[R_R] <- AGU_R[R_R] + AGU_N[N_R]; } {} {} |
                '(' R_R ')' '-'     {$$ = 0x08 | (R_R & 0x3); } {AGU_R[R_R]}
                   { AGU_R[R_R] <- AGU_R[R_R] - 1; }   {} {} |
                '(' R_R ')' '+'     {$$ = 0x0C | (R_R & 0x3); } {AGU_R[R_R]}
                   { AGU_R[R_R] <- AGU_R[R_R] + 1; }   {} {} |
```

```
                    ACCREGS  {$$ = 0x08 | ACCREGS; } {ACCREGS}{ACCREGS} {} {} |
                    R_R      {$$ = 0x10 | R_R; }  {AGU_R[R_R]}
                                                    {AGU_R[R_R]} {} {} |
                    N_R      {$$ = 0x18 | N_R; }  {AGU_N[N_R]}
                                                    {AGU_N[N_R]} {} {} |
                    M_R      {$$ = 0x20 | M_R; }  {AGU_M[M_R]}
                                                    {AGU_M[M_R]} {} {} |
                    PROGREGS {$$ = 0x38 | PROGREGS; }  {PROGREGS}
                                                    {PROGREGS} {} {} ;
Non_Terminal      SIMMD: SID INT {$$ = INT; } {INT} {INT} {} {} ;
Non_Terminal      LIMMD: LID INT {$$ = INT; } {INT} {INT} {} {} ;
Non_Terminal      SIMMA: SIA INT {$$ = INT; } {INT} {INT} {} {} ;
Non_Terminal      LIMMA: LIA INT {$$ = INT; } {INT} {INT} {} {} ;
Non_Terminal      ADRM:
                    '(' R_R ')' '-' N_R {$$ = 0x00 | (R_R & 0x7); }
                        { AGU_R[R_R] }
                            { AGU_R[R_R] <- AGU_R[R_R] - AGU_N[N_R]; } {} {} |
                    '(' R_R ')' '+' N_R {$$ = 0x08 | (R_R & 0x7); }
                        { AGU_R[R_R] }
                            { AGU_R[R_R] <- AGU_R[R_R] + AGU_N[N_R]; } {} {} |
                    '(' R_R ')' '-' {$$ = 0x10 | (R_R & 0x7); }
                        { AGU_R[R_R] } { AGU_R[R_R] <- AGU_R[R_R] - 1;} {} {} |
                    '(' R_R ')' '+' {$$ = 0x18 | (R_R & 0x7); }
                        { AGU_R[R_R] } { AGU_R[R_R] <- AGU_R[R_R] + 1;} {} {} |
                    '(' R_R ')' {$$ = 0x20 | (R_R & 0x7); }
                        { AGU_R[R_R] } {} {} {} |
                    '(' R_R '+' N_R ')' {$$ = 0x28 | (R_R & 0x7); }
                        { AGU_R[R_R] + AGU_N[N_R] } {}
                            { Cycle = 2; Size = 0; } {} |
                    '-' '(' R_R ')' {$$ = 0x38 | (R_R & 0x7); }
                        { AGU_R[R_R] - 1 } { AGU_R[R_R] <- AGU_R[R_R] - 1; }
                            { Cycle=2; Size=0; } {} ;
Non_Terminal      SEA:
                    '(' R_R ')' '-' N_R {$$ = 0x00 | (R_R & 0x7); }
                        { AGU_R[R_R] <- AGU_R[R_R] - AGU_N[N_R]; } {} {} {} |
                    '(' R_R ')' '+' N_R {$$ = 0x08 | (R_R & 0x7); }
                        { AGU_R[R_R] <- AGU_R[R_R] + AGU_N[N_R]; } {} {} {} |
                    '(' R_R ')' '-' {$$ = 0x10 | (R_R & 0x7); }
                        { AGU_R[R_R] <- AGU_R[R_R] - 1; } {} {} {} |
                    '(' R_R ')' '+' {$$ = 0x18 | (R_R & 0x7); }
                        { AGU_R[R_R] <- AGU_R[R_R] + 1; } {}
                            { Cycle = 2; Size = 0; } {} ;
Non_Terminal      INDEX:
                    R_R {$$ = R_R; } {AGU_R[R_R]} {AGU_R[R_R]} {} {} |
                    N_R {$$ = 0x8 | N_R; } {AGU_N[N_R]} {AGU_N[N_R]} {} {} ;
Non_Terminal       X_R:
                    X0_R {$$ = 0;} {X[0]} {X[0]} {} {} |
                    X1_R {$$ = 1;} {X[1]} {X[1]} {} {} ;
Non_Terminal       Y_R:
                    Y0_R {$$ = 0;} {Y[0]} {Y[0]} {} {} |
                    Y1_R {$$ = 1;} {Y[1]} {Y[1]} {} {} ;
RENAME(X0_R, X0_R1);
RENAME(Y0_R, Y0_R1);
Non_Terminal      MREG:
```

```
                  { AGU_R[R_R] + AGU_N[N_R] } {} { Cycle=2; Size=0; } {} |
                '-' '(' R_R ')' {$$ = 0x38 | (R_R & 0x7); } { AGU_R[R_R] - 1 }
                  { AGU_R[R_R] <- AGU_R[R_R] - 1; } { Cycle=2; Size=0;} {} |
                  ADDRESS {$$ = 0x30; Additional(0, Split.ADDR=ADDRESS;); }
                      {ADDRESS} {ADDRESS} { Cycle = 2; Size = 1; } {} ;
Non_Terminal      XYREGS:
                  X_R {$$ = 0x0 | X_R; } {X[X_R]} {X[X_R]} {} {} |
                  Y_R {$$ = 0x2 | Y_R; } {Y[Y_R]} {Y[Y_R]} {} {} ;
Non_Terminal      XYSRC:
                  X_D {$$ = 0; } {XREG} {XREG} {} {} |
                  Y_D {$$ = 1; } {YREG} {YREG} {} {} ;
Non_Terminal      ALUSRC:
                  A_D {$$ = 1; } {AREG} {AREG} {} {} |
                  B_D {$$ = 1; } {BREG} {BREG} {} {} |
                  X_D {$$ = 2; } {XREG} {XREG} {} {} |
                  Y_D {$$ = 3; } {YREG} {YREG} {} {} |
                  X_R {$$ = 4 + 2 * X_R; } {X[X_R]} {X[X_R]} {} {} |
                  Y_R {$$ = 5 + 2 * Y_R; } {Y[Y_R]} {Y[Y_R]} {} {} ;
Non_Terminal      REGS:
                  A_D {$$ = 0; } {AREG} {AREG} {} {} |
                  B_D {$$ = 0; } {BREG} {BREG} {} {} |
                  X_R {$$ = 4 | (X_R << 1); } {X[X_R]} {X[X_R]} {} {} |
                  Y_R {$$ = 5 | (Y_R << 1); } {Y[Y_R]} {Y[Y_R]} {} {} ;
Non_Terminal      ACCREGS:
                  A_R {$$ = (A_R << 1) | 0; } {A[A_R]} {A[A_R]} {} {} |
                  B_R {$$ = (B_R << 1) | 1; } {B[B_R]} {B[B_R]} {} {} |
                  A_D {$$ = 0x6; }      {AREG} {AREG} {} {} |
                  B_D {$$ = 0x7; }      {BREG} {BREG} {} {} ;
Non_Terminal      PROGREGS:
                  SR  {$$ = 0x1; }     {SRS}    {SRS} {} {} |
                  OMR {$$ = 0x2; }     {OMRS}   {OMRS} {} {} |
                  SP  {$$ = 0x3; }     {SPS}    {SPS} {} {} |
                  SSH {$$ = 0x4; }     {SSHS}   {SSHS} {} {} |
                  SSL {$$ = 0x5; }     {SSLS}   {SSLS} {} {} |
                  LA  {$$ = 0x6; }     {LAS}    {LAS} {} {} |
                  LC  {$$ = 0x7; }     {LCS}    {LCS} {} {} ;
Non_Terminal      ALUREGS:
                  XYREGS   {$$ = 0x04 | XYREGS; }  {XYREGS}  {XYREGS} {} {} |
                  ACCREGS  {$$ = 0x08 | ACCREGS; } {ACCREGS} {ACCREGS} {} {};
Non_Terminal      LAGREGS:
                  XYREGS   {$$ = 0x04 | XYREGS; }  {XYREGS}  {XYREGS} {} {} |
                  ACCREGS  {$$ = 0x08 | ACCREGS; } {ACCREGS} {ACCREGS} {} {}|
                  R_R      {$$ = 0x10 | R_R; }  {AGU_R[R_R]}
                                                  {AGU_R[R_R]} {} {} |
                  N_R      {$$ = 0x18 | N_R; }  {AGU_N[N_R]}
                                                  {AGU_N[N_R]} {} {} ;
Non_Terminal      LAGREGS2:
                  LAGREGS  {$$ = LAGREGS; }   {LAGREGS}   {LAGREGS} {} {} ;
Non_Terminal      PCREGS:
                  MR  {$$ = 0x0; }    {MRS}    {MRS} {} {} |
                  CCR {$$ = 0x1; }    {CCRS}   {CCRS} {} {} |
                  OMR {$$ = 0x2; }    {OMRS}   {OMRS} {} {} ;
Non_Terminal      DREGS:
                  XYREGS   {$$ = 0x04 | XYREGS; }  {XYREGS} {XYREGS} {} {} |
```

```
Token   CES             CES         { 0xD; };
Token   CLS             CLS         { 0xE; };
Token   CLE             CLE         { 0xF; };
Token   "-"             SIGN        { 0x1; };

//       type assembly        token   value
Non_Terminal      C_CODE:  CCC {$$ = CCC; } { CCRS[0] == 0 } {} {} {} |
                           CGE {$$ = CGE; } { (CCRS[3] ^ CCRS[1]) == 0 }
                                             {} {} {} |
                           CNE {$$ = CNE; } { CCRS[2] == 0 } {} {} {} |
                           CPL {$$ = CPL; } { CCRS[3] == 1 } {} {} {} |
                           CNN {$$ = CNN; } { (CCRS[2] |
                                             ((~CCRS[4])&(~CCRS[5])))
                                             == 0 } {} {} {} |
                           CEC {$$ = CEC; } { CCRS[5] == 0 } {} {} {} |
                           CLC {$$ = CLC; } { CCRS[6] == 0 } {} {} {} |
                           CGT {$$ = CGT; } { (CCRS[2] | (CCRS[3] ^ CCRS[1]))
                                             == 0 } {} {} {} |
                           CCS {$$ = CCS; } { CCRS[0] == 1 } {} {} {} |
                           CLT {$$ = CLT; } { (CCRS[3] ^ CCRS[1]) == 1 }
                                             {} {} {} |
                           CEQ {$$ = CEQ; } { CCRS[2] == 1 } {} {} {} |
                           CMI {$$ = CMI; } { CCRS[3] == 1 } {} {} {} |
                           CNR {$$ = CNR; } { (CCRS[2] |
                                             ((~CCRS[4])&(~CCRS[5])))
                                             == 1 } {} {} {} |
                           CES {$$ = CES; } { CCRS[5] == 1 } {} {} {} |
                           CLS {$$ = CLS; } { CCRS[6] == 0 } {} {} {} |
                           CLE {$$ = CLE; } { (CCRS[2] | (CCRS[3] ^ CCRS[1]))
                                             == 0 } {} {} {} ;
Non_Terminal      ADDRESS: INT {$$ = INT; } {INT} {INT} {} {} |
                           NAME {$$ = NAME;} {NAME} {NAME} {} {} ;
Non_Terminal      ACC: A_D {$$ = 0; } {AREG} {AREG} {} {} |
                       B_D {$$ = 1; } {BREG} {BREG} {} {} ;
Non_Terminal      ACCR:    A_D {$$ = 0; } {A} {A} {} {} |
                           B_D {$$ = 1; } {B} {B} {} {} ;
Non_Terminal      ACC2: ACC {$$ = ACC; } {ACC} {ACC} {} {} ;
Non_Terminal      SSR: X_R {$$ = X_R << 1; } {X[X_R]} {X[X_R]}  {} {}|
                       Y_R {$$ = 1 | (Y_R << 1); } {Y[Y_R]} {Y[Y_R]} {} {} ;
Non_Terminal      IMM: INT {$$ = INT; } {INT} {INT} {} {} ;
Non_Terminal      IMM2: INT {$$ = INT; } {INT} {INT}  {} {};
Non_Terminal      PPA: '<' '<' INT {$$ = INT & 0x3F; } {INT} {INT} {} {} ;
Non_Terminal      EA:
                 '(' R_R ')' '-' N_R {$$ = 0x00 | (R_R & 0x7); } { AGU_R[R_R] }
                    { AGU_R[R_R] <- AGU_R[R_R] - AGU_N[N_R]; } {} {} |
                 '(' R_R ')' '+' N_R {$$ = 0x08 | (R_R & 0x7); } { AGU_R[R_R] }
                    { AGU_R[R_R] <- AGU_R[R_R] + AGU_N[N_R]; } {} {} |
                 '(' R_R ')' '-' {$$ = 0x10 | (R_R & 0x7); } { AGU_R[R_R] }
                    { AGU_R[R_R] <- AGU_R[R_R] - 1; } {} {} |
                 '(' R_R ')' '+' {$$ = 0x18 | (R_R & 0x7); } { AGU_R[R_R] }
                    { AGU_R[R_R] <- AGU_R[R_R] + 1; } {} {} |
                 '(' R_R ')' {$$ = 0x20 | (R_R & 0x7); } { AGU_R[R_R] }
                    {} {} {} |
                 '(' R_R '+' N_R ')' {$$ = 0x28 | (R_R & 0x7); }
```

```
// ***** CHANGE, CLEARED, OVF, LIMIT, PLIMIT,
// etc need to be defined?

// This ends up in the lex file.
//       assembly        token  type     value
Token   X0              X0_R        {};
Token   X1              X1_R        {};
Token   Y0              Y0_R        {};
Token   Y1              Y1_R        {};
Token   A[0..2]         A_R         { [0..2]; };
Token   B[0..2]         B_R         { [0..2]; };
Token   R[0..7]         R_R         { [0..7]; };
Token   N[0..7]         N_R         { [0..7]; };
Token   M[0..7]         M_R         { [0..7]; };
Token   "A10"           A10_D       {};
Token   "B10"           B10_D       {};
Token   XREG            X_D         {};
Token   YREG            Y_D         {};
Token   AREG            A_D         {};
Token   BREG            B_D         {};
Token   LA              LA          {};
Token   LC              LC          {};
Token   SSH             SSH         {};
Token   SSL             SSL         {};
Token   SP              SP          {};
Token   PC              PC          {};
Token   MR              MR          {};
Token   CCR             CCR         {};
Token   SR              SR          {};
Token   OMR             OMR         {};
Token   SS              SS          {};
Token   AB              AB_D        {};
Token   BA              BA_D        {};
Token   "X:"            XMEM        {};
Token   "Y:"            YMEM        {};
Token   "L:"            LMEM        {};
Token   "P:"            PMEM        {};
Token   sia             SIA         {};
Token   lia             LIA         {};
Token   sid             SID         {};
Token   lid             LID         {};
Token   CCC             CCC         { 0x0; };
Token   CGE             CGE         { 0x1; };
Token   CNE             CNE         { 0x2; };
Token   CPL             CPL         { 0x3; };
Token   CNN             CNN         { 0x4; };
Token   CEC             CEC         { 0x5; };
Token   CLC             CLC         { 0x6; };
Token   CGT             CGT         { 0x7; };
Token   CCS             CCS         { 0x8; };
Token   CLT             CLT         { 0x9; };
Token   CEQ             CEQ         { 0xA; };
Token   CMI             CMI         { 0xB; };
Token   CNR             CNR         { 0xC; };
```

```
#define FALSE           0
#define DEFINE_NULL_OP  {} { NULLOP(); } {} {} {}

#define ADDCm(x,y,c,f)   ADDC(x,y,c,f,56,"sat")
#define SUBCm(x,y,c,f)   SUBC(x,y,c,f,56,"sat")
#define ADDm(x, y)       ADD(x,y,56,"sat")
#define SUBm(x, y)       SUB(x,y,56,"sat")
#define MULm(x,y)        MUL(x,y,56,24,"sat")
#define DIVm(x,y)        DIV(x,y,56,24,24,"sat")
#define ANDm(x,y)        AND(x,y,24)
#define XORm(x,y)        XOR(x,y,24)
#define ORm(x,y)         OR(x,y,24)
#define SORm(x,y)        OR(x,y,8)
#define NOTm(x)          NOT(x,24)
#define ANDSm(x,y)       AND(x,y,8)
#define ASLm(x)          ASL(x,1,0,CCRS[0],56)
#define ASRm(x)          ASR(x,1,x[55],CCRS[0],56)
#define LSLm(x)          ASL(x,1,0,CCRS[0],24)
#define LSRm(x)          ASR(x,1,0,CCRS[0],24)
#define ROLm(x)          ASL(x,1,CCRS[0],CCRS[0],24)
#define RORm(x)          ASR(x,1,CCRS[0],CCRS[0],24)
#define ABSm(x)          ABS(x,56)
#define RNDm(x)          RND(x, 56, 32, "upper")
// Shorthand names for flags
#define C_F          CCRS[0]
#define V_F          CCRS[1]
#define Z_F          CCRS[2]
#define N_F          CCRS[3]
#define U_F          CCRS[4]
#define E_F          CCRS[5]
#define L_F          CCRS[6]
#define LF_F         MRS[8]
#define T_F          MRS[6]
#define SET_FLAG(c,f)   if (c) { f <- 1;} else {f <- 0;}
#define SET_ALL(v,z,n,u,e)                                  \
                        SET_FLAG(v,V_F);                    \
                        SET_FLAG(z,Z_F);                    \
                        SET_FLAG(n,N_F);                    \
                        SET_FLAG(u,U_F);                    \
                        SET_FLAG(e,E_F);
#define CLEAR_ALL(v,z,n,u,e)                                \
    V_F <- v; Z_F <- z; N_F <- n; U_F <- u; E_F <- e;

#define SIGNED(acc)     (((S_MODE == 0) &                                   \
                          (((acc >> 47) == 0x0) | ((acc >> 47) == 0x1FF))) |  \
                         ((S_MODE == 1) &                                   \
                          (((acc >> 48) == 0x0) | ((acc >> 48) == 0xFF))) |   \
                         ((S_MODE == 2) &                                   \
                          (((acc >> 46) == 0x0) | ((acc >> 46) == 0x3FF))))

#define UNORM(acc)      (((S_MODE == 0) & !(acc[47] ^ acc[46])) |           \
                         ((S_MODE == 1) & !(acc[48] ^ acc[47])) |           \
                         ((S_MODE == 1) & !(acc[46] ^ acc[45])))
```

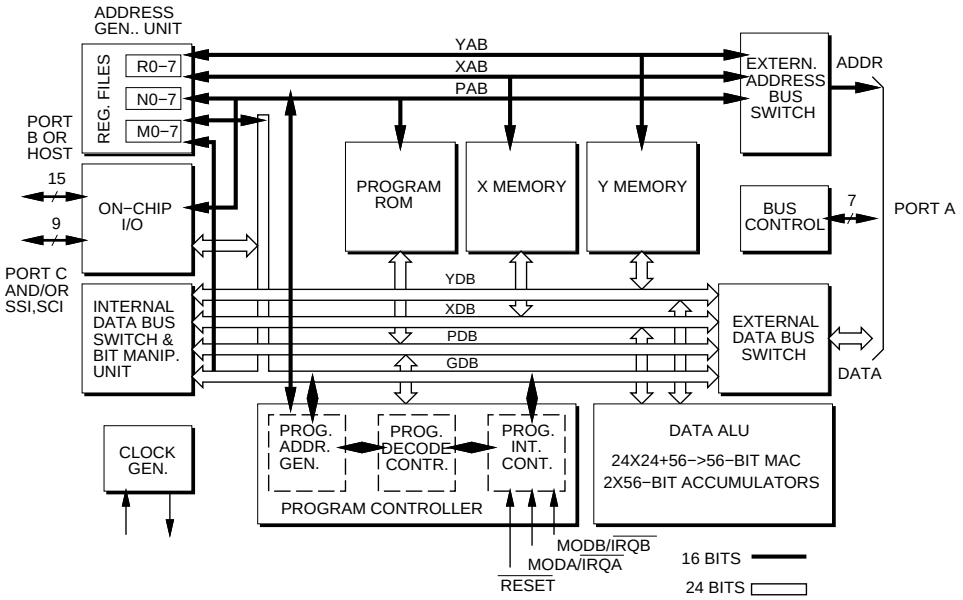

Figure 9: **The Motorola 56000 DSP engine.**

C.2 The Motorola 56000 DSP

```
// Conventions:

// | = OR
// & = AND
// [x..y] = range from x to y
// , = used between required fields
// @ = variable declaration follows

// \ = ignore special symbol


// ---------------------------------------------------------------------------
// Number of fields in each VLIW word
// Size and format of each

Section Format

DBM     = OP[8], MODE[8];
Main    = OP[8];


// ---------------------------------------------------------------------------
Section Global_Definitions


// Some Macro definitions
//
#define RENAME(x,y) Non_Terminal y: x { $$ = x; } { x } { x } { } { }

#define TRUE            1
```

```
~((AG?_add* *,*,@[1]) & (DB_move *,@[1]))
~((AG?_inc *,*,@[1]) & (DB_move *,@[1]))
~((MAC_*w *,*,@[1]) & (DB_move *,@[1]))
~((ALU_* *,*,@[1]) & (DB_move *,@[1]))

// --------------------------------------------------------------------------

Section Optional
```

```
// bitfield conflict on ALU.RMEM
~((ALU_*c *) & ((DB_move *,ALU.R*) | (DB_move ALU.R*,*)))

// SRC and SINK cannot be the same in a data bus transfer
// units connected to the bus only have one port to it each
~(DB_move @[1].*,@[1].*)

// Cannot drive both memories on the bus at the same time.
// Bus conflict
~((DM1_bus_load* *) & (DM2_bus_load* *))

// Also cannot read both memories from the bus at the same time
// bitfiled conflict on DB.SINK
~((DM1_bus_save* *) & (DM2_bus_save* *))

// Cannot do a memory bus transfer and a normal bus transfer at the same time
~((DM?_bus* *) & (DB_move *))

// If you are doing an AG* register update with a constant and a memory
// access with a constant make sure the constants are the same - they share
// the same field AG*.RMEM
(~((DM1_*c *,@[1]) & (AG1_add?c *,@[1],*)) |
  ((DM1_*c *,@[1]) & (AG1_add?c *,@[1],*)) )


// If Instruction Memory (DATA) is the source then can't use AG1 or AG2
// or data memory transfers that use the address generators or transfer to
// an address generator file
// Bit conflict on all the fields of AG1 and AG2
~((DB_move DATA, *) &
  ((((AG?_* *)      |
   (DM?_bus* *))     |
   (DM?_dir_*_i*))   |
   (DB_move DATA, AG*)))

// load jump register and branch instructions can't be done with
// AG2 instructions, DM2 operations that use the AG2 or a transfer to or from
// the AG2 regfile
// Bitfield conflict on the AG2 subfields
~(((((Control_ldjr *)        |
   (Control_br *))           |
   (Control_brcz *))         |
   (Control_brcp *))
&
  (((((AG2_* *)              |
   (DM2_bus* *))             |
   (DM2_dir_*_i*))           |
   (DB_move *,AG2.R*))       |
   (DB_move AG2.R*,*)))

// Can't write to same registers from both execution unit operation and
// DB or transfer path
// Example: If ALU write back to ALU.R1 then DB_move *,ALU.R1 is not allowed.
// conflict at the register storage set lines
```

```
        DM2_bus_load_i      SINK, AG2_RA                { DB.SRC = 0x1;
                                                          DM2A.OP = 0x0;
                                                          AG2.RA = AG2_RA;
                                                          AG2.OP=AG2.OP+0x0;}
                    { SINK <- DM2[AG2_RA] ; }
                    {}
                    { Cycle=1; Size=1; Stall=0; }
                    { Latency=1; Usage=1; }

        DM2_bus_load_io     SINK, AG2_RA, AG2_RB        { DB.SRC = 0x1;
                                                          DM2A.OP = 0x0;
                                                          AG2.RA = AG2_RA;
                                                          AG2.RB = AG2_RB;
                                                          AG2.OP=AG2.OP+0x5;}
                    { SINK <- DM2[ ADDm(AG2_RA, AG2_RB)] ; }
                    {}
                    { Cycle=1; Size=1; Stall=0; }
                    { Latency=1; Usage=1; }

        DM2_bus_load_ic     SINK, AG2_RA, CONST         { DB.SRC = 0x1;
                                                          DM2A.OP = 0x0;
                                                          AG2.RA = AG2_RA;
                                                          AG2.RMEM = CONST;
                                                          AG2.OP=AG2.OP+0xA;}
                    { SINK <- DM2[ ADDm(AG2_RA, ext3(CONST))] ; }
                    {}
                    { Cycle=1; Size=1; Stall=0; }
                    { Latency=1; Usage=1; }

        DM2_idle                                        { MACOP.OP = 0x2;
                                                          DM2A.OP = 0x1; }
                    {}
                    {}
                    { Cycle=1; Size=1; Stall=0; }
                    { Latency=1; Usage=1; }

// ------------------------------------------------------------------------
// Any restrictions (in the form of rules) of which instructions go together
// and which do not.

Section Constraints

// Can't do constant addressing memory mode with AG transfers on bus
// bitfield conflict on AG*.RMEM
~((DM1_*ic *) & ((DB_move *,AG1.R*) | (DB_move AG1.R*,*)))
~((DM2_*ic *) & ((DB_move *,AG2.R*) | (DB_move AG2.R*,*)))

// Can't do register updates with constant in AG with AG bus transfers
// bitfield conflict on AG*.RMEM
~((AG1_add?c *) & ((DB_move *,AG1.R*) | (DB_move AG1.R*,*)))
~((AG2_add?c *) & ((DB_move *,AG2.R*) | (DB_move AG2.R*,*)))

// Can't do ALU operations involving constants with ALU bus transfers
```

```
                                                AG2.OP=AG2.OP+0x5;}
            { MAC_RMEM <- DM2[ ADDm(AG2_RA, AG2_RB)] ; }
            {}
            { Cycle=1; Size=1; Stall=0; }
            { Latency=1; Usage=1; }

DM2_dir_load_ic     MAC_RMEM, AG2_RA, CONST     { MAC.RMEM = MAC_RMEM;
                                                  MACOP.OP = 0x1;
                                                  DM2A.OP = 0x0;
                                                  AG2.RA = AG2_RA;
                                                  AG2.RMEM = CONST;
                                                  AG2.OP=AG2.OP+0xA;}
            { MAC_RMEM <- DM2[ ADDm(AG2_RA, ext3(CONST))] ; }
            {}
            { Cycle=1; Size=1; Stall=0; }
            { Latency=1; Usage=1; }

DM2_dir_load_m      MAC_RMEM, SRC               { MAC.RMEM = MAC_RMEM;
                                                  MACOP.OP = 0x1;
                                                  DM2A.OP = 0x1;
                                                  DB.SINK = 0xB; }
            { MAC_RMEM <- DM2[SRC] ; }
            {}
            { Cycle=1; Size=1; Stall=0; }
            { Latency=1; Usage=1; }

DM2_bus_save_i      SRC, AG2_RA                 { DB.SINK = 0x1;
                                                  DM2A.OP = 0x0;
                                                  AG2.RA = AG2_RA;
                                                  AG2.OP=AG2.OP+0x0;}
            { DM2[AG2_RA] <- SRC; }
            {}
            { Cycle=1; Size=1; Stall=0; }
            { Latency=1; Usage=1; }

DM2_bus_save_io     SRC, AG2_RA, AG2_RB         { DB.SINK = 0x1;
                                                  DM2A.OP = 0x0;
                                                  AG2.RA = AG2_RA;
                                                  AG2.RB = AG2_RB;
                                                  AG2.OP=AG2.OP+0x5;}
            { DM2[ ADDm(AG2_RA, AG2_RB)] <- SRC; }
            {}
            { Cycle=1; Size=1; Stall=0; }
            { Latency=1; Usage=1; }

DM2_bus_save_ic     SRC, AG2_RA, CONST          { DB.SINK = 0x1;
                                                  DM2A.OP = 0x0;
                                                  AG2.RA = AG2_RA;
                                                  AG2.RMEM = CONST;
                                                  AG2.OP=AG2.OP+0xA;}
            { DM2[ ADDm(AG2_RA, ext3(CONST))] <- SRC; }
            {}
            { Cycle=1; Size=1; Stall=0; }
            { Latency=1; Usage=1; }
```

```
                                                DM2A.OP = 0x0;
                                                AG2.RA = AG2_RA;
                                                AG2.OP=AG2.OP+0x0;}
            { DM2[AG2_RA] <- MAC_RMEM; }
            {}
            { Cycle=1; Size=1; Stall=0; }
            { Latency=1; Usage=1; }

DM2_dir_save_io     MAC_RMEM, AG2_RA, AG2_RB    { MAC.RMEM = MAC_RMEM;
                                                  MACOP.OP = 0x0;
                                                  DM2A.OP = 0x0;
                                                  AG2.RA = AG2_RA;
                                                  AG2.RB = AG2_RB;
                                                  AG2.OP=AG2.OP+0x5;}
            { DM2[ ADDm(AG2_RA, AG2_RB)] <- MAC_RMEM; }
            {}
            { Cycle=1; Size=1; Stall=0; }
            { Latency=1; Usage=1; }

DM2_dir_save_ic     MAC_RMEM, AG2_RA, CONST     { MAC.RMEM = MAC_RMEM;
                                                  MACOP.OP = 0x0;
                                                  DM2A.OP = 0x0;
                                                  AG2.RA = AG2_RA;
                                                  AG2.RMEM = CONST;
                                                  AG2.OP=AG2.OP+0xA;}
            { DM2[ ADDm(AG2_RA, ext3(CONST))] <- MAC_RMEM; }
            {}
            { Cycle=1; Size=1; Stall=0; }
            { Latency=1; Usage=1; }

DM2_dir_save_m      MAC_RMEM, SRC               { MAC.RMEM = MAC_RMEM;
                                                  MACOP.OP = 0x0;
                                                  DM2A.OP = 0x1;
                                                  DB.SINK = 0xB; }
            { DM2[SRC] <- MAC_RMEM; }
            {}
            { Cycle=1; Size=1; Stall=0; }
            { Latency=1; Usage=1; }

DM2_dir_load_i      MAC_RMEM, AG2_RA            { MAC.RMEM = MAC_RMEM;
                                                  MACOP.OP = 0x1;
                                                  DM2A.OP = 0x0;
                                                  AG2.RA = AG2_RA;
                                                  AG2.OP=AG2.OP+0x0;}
            { MAC_RMEM <- DM2[AG2_RA] ; }
            {}
            { Cycle=1; Size=1; Stall=0; }
            { Latency=1; Usage=1; }

DM2_dir_load_io     MAC_RMEM, AG2_RA, AG2_RB    { MAC.RMEM = MAC_RMEM;
                                                  MACOP.OP = 0x1;
                                                  DM2A.OP = 0x0;
                                                  AG2.RA = AG2_RA;
                                                  AG2.RB = AG2_RB;
```

```
        DM1_bus_save_ic     SRC, AG1_RA, CONST          { DB.SINK = 0x0;
                                                          DM1A.OP = 0x0;
                                                          AG1.RA = AG1_RA;
                                                          AG1.RMEM = CONST;
                                                          AG1.OP=AG1.OP+0xA;}
                    { DM1[ ADDm(AG1_RA, ext3(CONST))] <- SRC; }
                    {}
                    { Cycle=1; Size=1; Stall=0; }
                    { Latency=1; Usage=1; }

        DM1_bus_load_i      SINK, AG1_RA                { DB.SRC = 0x0;
                                                          DM1A.OP = 0x0;
                                                          AG1.RA = AG1_RA;
                                                          AG1.OP=AG1.OP+0x0;}
                    { SINK <- DM1[AG1_RA] ; }
                    {}
                    { Cycle=1; Size=1; Stall=0; }
                    { Latency=1; Usage=1; }

        DM1_bus_load_io     SINK, AG1_RA, AG1_RB        { DB.SRC = 0x0;
                                                          DM1A.OP = 0x0;
                                                          AG1.RA = AG1_RA;
                                                          AG1.RB = AG1_RB;
                                                          AG1.OP=AG1.OP+0x5;}
                    { SINK <- DM1[ ADDm(AG1_RA, AG1_RB)] ; }
                    {}
                    { Cycle=1; Size=1; Stall=0; }
                    { Latency=1; Usage=1; }

        DM1_bus_load_ic     SINK, AG1_RA, CONST         { DB.SRC = 0x0;
                                                          DM1A.OP = 0x0;
                                                          AG1.RA = AG1_RA;
                                                          AG1.RMEM = CONST;
                                                          AG1.OP=AG1.OP+0xA;}
                    { SINK <- DM1[ ADDm(AG1_RA, ext3(CONST))] ; }
                    {}
                    { Cycle=1; Size=1; Stall=0; }
                    { Latency=1; Usage=1; }

        DM1_idle                                        { ALUOP.OP = 0x2;
                                                          DM1A.OP = 0x1; }
                    {}
                    {}
                    { Cycle=1; Size=1; Stall=0; }
                    { Latency=1; Usage=1; }

// ////////////////////////////////////////////////////////////////////////
// Data Memory 2 operations

Field DM2f:
        DM2_NULL                                        DEFINE_NULL_OP
        DM2_dir_save_i      MAC_RMEM, AG2_RA            { MAC.RMEM = MAC_RMEM;
                                                          MACOP.OP = 0x0;
```

```
            { ALU_RMEM <- DM1[AG1_RA] ; }
            {}
            { Cycle=1; Size=1; Stall=0; }
            { Latency=1; Usage=1; }

DM1_dir_load_io     ALU_RMEM, AG1_RA, AG1_RB    { ALU.RMEM = ALU_RMEM;
                                                  ALUOP.OP = 0x1;
                                                  DM1A.OP = 0x0;
                                                  AG1.RA = AG1_RA;
                                                  AG1.RB = AG1_RB;
                                                  AG1.OP=AG1.OP+0x5;}
            { ALU_RMEM <- DM1[ ADDm(AG1_RA, AG1_RB)] ; }
            {}
            { Cycle=1; Size=1; Stall=0; }
            { Latency=1; Usage=1; }

DM1_dir_load_ic     ALU_RMEM, AG1_RA, CONST     { ALU.RMEM = ALU_RMEM;
                                                  ALUOP.OP = 0x1;
                                                  DM1A.OP = 0x0;
                                                  AG1.RA = AG1_RA;
                                                  AG1.RMEM = CONST;
                                                  AG1.OP=AG1.OP+0xA;}
            { ALU_RMEM <- DM1[ ADDm(AG1_RA, ext3(CONST))] ; }
            {}
            { Cycle=1; Size=1; Stall=0; }
            { Latency=1; Usage=1; }

DM1_dir_load_m      ALU_RMEM, SRC              { ALU.RMEM = ALU_RMEM;
                                                  ALUOP.OP = 0x1;
                                                  DM1A.OP = 0x1;
                                                  DB.SINK = 0xA; }
            { ALU_RMEM <- DM1[SRC] ; }
            {}
            { Cycle=1; Size=1; Stall=0; }
            { Latency=1; Usage=1; }

DM1_bus_save_i      SRC, AG1_RA                { DB.SINK = 0x0;
                                                  DM1A.OP = 0x0;
                                                  AG1.RA = AG1_RA;
                                                  AG1.OP=AG1.OP+0x0;}
            { DM1[AG1_RA] <- SRC; }
            {}
            { Cycle=1; Size=1; Stall=0; }
            { Latency=1; Usage=1; }

DM1_bus_save_io     SRC, AG1_RA, AG1_RB        { DB.SINK = 0x0;
                                                  DM1A.OP = 0x0;
                                                  AG1.RA = AG1_RA;
                                                  AG1.RB = AG1_RB;
                                                  AG1.OP=AG1.OP+0x5;}
            { DM1[ ADDm(AG1_RA, AG1_RB)] <- SRC; }
            {}
            { Cycle=1; Size=1; Stall=0; }
            { Latency=1; Usage=1; }
```

```
                                { Cycle=1; Size=1; Stall=0; }
                                { Latency=1; Usage=1; }

// ////////////////////////////////////////////////////////////////////////
// Data Memory 1 operations

Field DM1f:
        DM1_NULL                                            DEFINE_NULL_OP
        DM1_dir_save_i      ALU_RMEM, AG1_RA                { ALU.RMEM = ALU_RMEM;
                                                              ALUOP.OP = 0x0;
                                                              DM1A.OP = 0x0;
                                                              AG1.RA = AG1_RA;
                                                              AG1.OP=AG1.OP+0x0;}
                    { DM1[AG1_RA] <- ALU_RMEM; }
                    {}
                    { Cycle=1; Size=1; Stall=0; }
                    { Latency=1; Usage=1; }

        DM1_dir_save_io     ALU_RMEM, AG1_RA, AG1_RB        { ALU.RMEM = ALU_RMEM;
                                                              ALUOP.OP = 0x0;
                                                              DM1A.OP = 0x0;
                                                              AG1.RA = AG1_RA;
                                                              AG1.RB = AG1_RB;
                                                              AG1.OP=AG1.OP+0x5;}
                    { DM1[ ADDm(AG1_RA, AG1_RB)] <- ALU_RMEM; }
                    {}
                    { Cycle=1; Size=1; Stall=0; }
                    { Latency=1; Usage=1; }

        DM1_dir_save_ic     ALU_RMEM, AG1_RA, CONST         { ALU.RMEM = ALU_RMEM;
                                                              ALUOP.OP = 0x0;
                                                              DM1A.OP = 0x0;
                                                              AG1.RA = AG1_RA;
                                                              AG1.RMEM = CONST;
                                                              AG1.OP=AG1.OP+0xA;}
                    { DM1[ ADDm(AG1_RA, ext3(CONST))] <- ALU_RMEM; }
                    {}
                    { Cycle=1; Size=1; Stall=0; }
                    { Latency=1; Usage=1; }

        DM1_dir_save_m      ALU_RMEM, SRC                   { ALU.RMEM = ALU_RMEM;
                                                              ALUOP.OP = 0x0;
                                                              DM1A.OP = 0x1;
                                                              DB.SINK = 0xA; }
                    { DM1[SRC] <- ALU_RMEM; }
                    {}
                    { Cycle=1; Size=1; Stall=0; }
                    { Latency=1; Usage=1; }

        DM1_dir_load_i      ALU_RMEM, AG1_RA                { ALU.RMEM = ALU_RMEM;
                                                              ALUOP.OP = 0x1;
                                                              DM1A.OP = 0x0;
                                                              AG1.RA = AG1_RA;
                                                              AG1.OP=AG1.OP+0x0;}
```

```
                              { Latency=1; Usage=1; }
        AG1_add         AG1_RA,AG1_RB,AG1_RW            { AG1.OP = 0x3 ;
                                                          AG1.RA = AG1_RA ;
                                                          AG1.RB = AG1_RB ;
                                                          AG1.RW = AG1_RW ; }
                             { AG1_RW <-  ADDm(AG1_RB,AG1_RA) ; }
                              {}
                              { Cycle=1; Size=1; Stall=0; }
                              { Latency=1; Usage=1; }
        AG1_idle                                        { AG1.OP = 0x4 ; }
                              {}
                              {}
                              { Cycle=1; Size=1; Stall=0; }
                              { Latency=1; Usage=1; }

// ////////////////////////////////////////////////////////////////////////
// Address Generator 2 register updates

Field AG2f:
        AG2_NULL                                        DEFINE_NULL_OP
        AG2_addbc       AG2_RB,CONST,AG2_RW             { AG2.OP = 0x0 ;
                                                          AG2.RB = AG2_RB ;
                                                          AG2.RMEM = CONST ;
                                                          AG2.RW = AG2_RW ; }
                 { AG2_RW <-  ADDm(AG2_RB, ext3(CONST)) ; }
                              {}
                              { Cycle=1; Size=1; Stall=0; }
                              { Latency=1; Usage=1; }
        AG2_addac       AG2_RA,CONST,AG2_RW             { AG2.OP = 0x1 ;
                                                          AG2.RA = AG2_RA ;
                                                          AG2.RMEM = CONST ;
                                                          AG2.RW = AG2_RW ; }
                 { AG2_RW <-  ADDm(AG2_RA, ext3(CONST)) ; }
                              {}
                              { Cycle=1; Size=1; Stall=0; }
                              { Latency=1; Usage=1; }
        AG2_inc         AG2_RB,AG2_RW                   { AG2.OP = 0x2 ;
                                                          AG2.RB = AG2_RB ;
                                                          AG2.RW = AG2_RW ; }
                 { AG2_RW <-  ADDm(AG2_RB, 0x00000001) ; }
                              {}
                              { Cycle=1; Size=1; Stall=0; }
                              { Latency=1; Usage=1; }
        AG2_add         AG2_RA,AG2_RB,AG2_RW            { AG2.OP = 0x3 ;
                                                          AG2.RA = AG2_RA ;
                                                          AG2.RB = AG2_RB ;
                                                          AG2.RW = AG2_RW ; }
                             { AG2_RW <-  ADDm(AG2_RB,AG2_RA) ; }
                              {}
                              { Cycle=1; Size=1; Stall=0; }
                              { Latency=1; Usage=1; }
        AG2_idle                                        { AG2.OP = 0x4 ; }
                              {}
                              {}
```

```
                                  { ACC <- FMULm(MAC_RA, MAC_RB) ;
                                   MAC_RW <- FMULm(MAC_RA, MAC_RB) ; }
                                 {}
                                 { Cycle=1; Size=1; Stall=0; }
                                 { Latency=1; Usage=1; }
        MAC_lda     MAC_RA                      { MAC.OP = 0x6 ;
                                                  MAC.RA = MAC_RA ; }
                                 { ACC <- MAC_RA ; }
                                 {}
                                 { cycle=1; size=1; stall=0; }
                                 { latency=1; Usage=1; }
        MAC_clr                  { MAC.OP = 0x7 ; }
                                 { ACC <- 0x00000000 ; }
                                 {}
                                 { cycle=1; size=1; stall=0; }
                                 { latency=1; Usage=1; }


// ////////////////////////////////////////////////////////////////////////

Field DB:
        DB_NULL               DEFINE_NULL_OP
        DB_move    SRC,SINK   { } { SINK <- SRC ; }
                                 {}
                                 { Cycle=1; Size=1; Stall=0; }
                                 { Latency=1; Usage=1; }

// ////////////////////////////////////////////////////////////////////////
// Address Generator 1 register updates

Field AG1f:
        AG1_NULL                                          DEFINE_NULL_OP
        AG1_addbc      AG1_RB,CONST,AG1_RW                { AG1.OP = 0x0 ;
                                                            AG1.RB = AG1_RB ;
                                                            AG1.RMEM = CONST ;
                                                            AG1.RW = AG1_RW ; }
                                 { AG1_RW <-  ADDm(AG1_RB, ext3(CONST)) ; }
                                 {}
                                 { Cycle=1; Size=1; Stall=0; }
                                 { Latency=1; Usage=1; }
        AG1_addac      AG1_RA,CONST,AG1_RW                { AG1.OP = 0x1 ;
                                                            AG1.RA = AG1_RA ;
                                                            AG1.RMEM = CONST ;
                                                            AG1.RW = AG1_RW ; }
                 { AG1_RW <-  ADDm(AG1_RA, ext3(CONST)) ; }
                                 {}
                                 { Cycle=1; Size=1; Stall=0; }
                                 { Latency=1; Usage=1; }
        AG1_inc        AG1_RB,AG1_RW                      { AG1.OP = 0x2 ;
                                                            AG1.RB = AG1_RB ;
                                                            AG1.RW = AG1_RW ; }
                 { AG1_RW <-  ADDm(AG1_RB, 0x00000001) ; }
                                 {}
                                 { Cycle=1; Size=1; Stall=0; }
```

```
                                        ALU.RW = ALU_RW ; }
                        { ALU_RW <-  LSRm(ALU_RA, ext10(CONST)) ; }
                                 {}
                                 { Cycle=1; Size=1; Stall=0; }
                                 { Latency=1; Usage=1; }


// ////////////////////////////////////////////////////////////////////////

Field MACf:
        MAC_NULL                                DEFINE_NULL_OP
        MAC_mac     MAC_RA,MAC_RB               { MAC.OP = 0x0 ;
                                                  MAC.RA = MAC_RA ;
                                                  MAC.RB = MAC_RB ; }
                           { ACC <- FMULm(MAC_RA, FADDm(MAC_RB, ACC)) ; }
                                 {}
                                 { Cycle=1; Size=1; Stall=0; }
                                 { Latency=1; Usage=1; }
        MAC_macw    MAC_RA,MAC_RB,MAC_RW        { MAC.OP = 0x1 ;
                                                  MAC.RA = MAC_RA ;
                                                  MAC.RB = MAC_RB ;
                                                  MAC.RW = MAC_RW ; }
                           { ACC <- FMULm(MAC_RA, FADDm(MAC_RB, ACC)) ;
                          MAC_RW <- FMULm(MAC_RA, FADDm(MAC_RB, ACC)) ; }
                                 {}
                                 { Cycle=1; Size=1; Stall=0; }
                                 { Latency=1; Usage=1; }
        MAC_add     MAC_RA,MAC_RB               { MAC.OP = 0x2 ;
                                                  MAC.RA = MAC_RA ;
                                                  MAC.RB = MAC_RB ; }
                                   { ACC <- FADDm(MAC_RA, MAC_RB) ; }
                                 {}
                                 { Cycle=1; Size=1; Stall=0; }
                                 { Latency=1; Usage=1; }
        MAC_addw    MAC_RA,MAC_RB,MAC_RW        { MAC.OP = 0x3 ;
                                                  MAC.RA = MAC_RA ;
                                                  MAC.RB = MAC_RB ;
                                                  MAC.RW = MAC_RW ; }
                                   { ACC <- FADDm(MAC_RA, MAC_RB) ;
                                  MAC_RW <- FADDm(MAC_RA, MAC_RB) ; }
                                 {}
                                 { Cycle=1; Size=1; Stall=0; }
                                 { Latency=1; Usage=1; }
        MAC_mul     MAC_RA,MAC_RB               { MAC.OP = 0x4 ;
                                                  MAC.RA = MAC_RA ;
                                                  MAC.RB = MAC_RB ; }
                                   { ACC <- FMULm(MAC_RA, MAC_RB) ; }
                                 {}
                                 { Cycle=1; Size=1; Stall=0; }
                                 { Latency=1; Usage=1; }
        MAC_mulw    MAC_RA,MAC_RB,MAC_RW        { MAC.OP = 0x5 ;
                                                  MAC.RA = MAC_RA ;
                                                  MAC.RB = MAC_RB ;
                                                  MAC.RW = MAC_RW ; }
```

```
                                        ALU.RW = ALU_RW ; }
                          { ALU_RW <-  IORm(ALU_RA, ALU_RB) ; }
                          {}
                          { Cycle=1; Size=1; Stall=0; }
                          { Latency=1; Usage=1; }
ALU_not     ALU_RA,ALU_RW             { ALU.OP = 0x9 ;
                                        ALU.RA = ALU_RA ;
                                        ALU.RW = ALU_RW ; }
                                 { ALU_RW <-  NOTm(ALU_RA) ; }
                          {}
                          { Cycle=1; Size=1; Stall=0; }
                          { Latency=1; Usage=1; }
ALU_xor     ALU_RA,ALU_RB,ALU_RW      { ALU.OP = 0xA ;
                                        ALU.RA = ALU_RA ;
                                        ALU.RB = ALU_RB ;
                                        ALU.RW = ALU_RW ; }
                          { ALU_RW <-  XORm(ALU_RA, ALU_RB) ; }
                          {}
                          { Cycle=1; Size=1; Stall=0; }
                          { Latency=1; Usage=1; }
ALU_pass    ALU_RA,ALU_RW             { ALU.OP = 0xB ;
                                        ALU.RA = ALU_RA ;
                                        ALU.RW = ALU_RW ; }
                                   { ALU_RW <- ALU_RA ; }
                          {}
                          { Cycle=1; Size=1; Stall=0; }
                          { Latency=1; Usage=1; }
ALU_lsl     ALU_RA,ALU_RB,ALU_RW      { ALU.OP = 0xC ;
                                        ALU.RA = ALU_RA ;
                                        ALU.RB = ALU_RB ;
                                        ALU.RW = ALU_RW ; }
                        { ALU_RW <-  LSLm(ALU_RA, ALU_RB) ; }
                          {}
                          { Cycle=1; Size=1; Stall=0; }
                          { Latency=1; Usage=1; }
ALU_lslc    ALU_RA,CONST,ALU_RW       { ALU.OP = 0xD ;
                                        ALU.RA = ALU_RA ;
                                        Split.CONSTs = CONST ;
                                        ALU.RW = ALU_RW ; }
               { ALU_RW <-  LSLm(ALU_RA, ext10(CONST)) ; }
                          {}
                          { Cycle=1; Size=1; Stall=0; }
                          { Latency=1; Usage=1; }
ALU_lsr     ALU_RA,ALU_RB,ALU_RW      { ALU.OP = 0xE ;
                                        ALU.RA = ALU_RA ;
                                        ALU.RB = ALU_RB ;
                                        ALU.RW = ALU_RW ; }
                        { ALU_RW <-  LSRm(ALU_RA, ALU_RB) ; }
                          {}
                          { Cycle=1; Size=1; Stall=0; }
                          { Latency=1; Usage=1; }
ALU_lsrc    ALU_RA,CONST,ALU_RW       { ALU.OP = 0xF ;
                                        ALU.RA = ALU_RA ;
                                        Split.CONSTs = CONST ;
```

```
                        {}
                        { Cycle=1; Size=1; Stall=0; }
                        { Latency=1; Usage=1; }
ALU_sub     ALU_RA,ALU_RB,ALU_RW      { ALU.OP = 0x2 ;
                                        ALU.RA = ALU_RA ;
                                        ALU.RB = ALU_RB ;
                                        ALU.RW = ALU_RW ; }
                        { ALU_RW <-  SUBm(ALU_RA, ALU_RB) ; }
                        {}
                        { Cycle=1; Size=1; Stall=0; }
                        { Latency=1; Usage=1; }
ALU_subc    ALU_RA,CONST,ALU_RW       { ALU.OP = 0x3 ;
                                        ALU.RA = ALU_RA ;
                                        Split.CONSTs = CONST ;
                                        ALU.RW = ALU_RW ; }
                { ALU_RW <-  SUBm(ALU_RA, sx10(CONST)) ; }
                        {}
                        { Cycle=1; Size=1; Stall=0; }
                        { Latency=1; Usage=1; }
ALU_mul     ALU_RA,ALU_RB,ALU_RW      { ALU.OP = 0x4 ;
                                        ALU.RA = ALU_RA ;
                                        ALU.RB = ALU_RB ;
                                        ALU.RW = ALU_RW ; }
                        { ALU_RW <-  MULm(ALU_RA, ALU_RB) ; }
                        {}
                        { Cycle=1; Size=1; Stall=0; }
                        { Latency=1; Usage=1; }
ALU_mulc    ALU_RA,CONST,ALU_RW       { ALU.OP = 0x5 ;
                                        ALU.RA = ALU_RA ;
                                        Split.CONSTs = CONST ;
                                        ALU.RW = ALU_RW ; }
                { ALU_RW <-  MULm(ALU_RA, sx10(CONST)) ; }
                        {}
                        { Cycle=1; Size=1; Stall=0; }
                        { Latency=1; Usage=1; }
ALU_div     ALU_RA,ALU_RB,ALU_RW      { ALU.OP = 0x6 ;
                                        ALU.RA = ALU_RA ;
                                        ALU.RB = ALU_RB ;
                                        ALU.RW = ALU_RW ; }
                        { ALU_RW <-  DIVm(ALU_RA, ALU_RB) ; }
                        {}
                        { Cycle=1; Size=1; Stall=0; }
                        { Latency=1; Usage=1; }
ALU_and     ALU_RA,ALU_RB,ALU_RW      { ALU.OP = 0x7 ;
                                        ALU.RA = ALU_RA ;
                                        ALU.RB = ALU_RB ;
                                        ALU.RW = ALU_RW ; }
                        { ALU_RW <-  ANDm(ALU_RA, ALU_RB) ; }
                        {}
                        { Cycle=1; Size=1; Stall=0; }
                        { Latency=1; Usage=1; }
ALU_or      ALU_RA,ALU_RB,ALU_RW      { ALU.OP = 0x8 ;
                                        ALU.RA = ALU_RA ;
                                        ALU.RB = ALU_RB ;
```

```
                            { Latency=1; Usage=1; }
        Control_brcz RI,OFFSET  { Control.OP = 0xA ;
                                  Control.RI = RI ;
                                  Split.OFFSETs = OFFSET ; }
                                { if (RI == 0)
                                    { PC <- ADDm(PC, sx16(OFFSET)) ; }
                                  else
                                    { PC <-  PC + 1 ; } ; }
                                {}
                                { Cycle=1; Size=1; Stall=0; }
                                { Latency=1; Usage=1; }
        Control_brcp RI,OFFSET  { Control.OP = 0xB ;
                                  Control.RI = RI ;
                                  Split.OFFSETs = OFFSET ; }
                                { if (RI > 0)
                                    { PC <-  ADDm(PC, sx16(OFFSET)) ; }; }
                                {}
                                { Cycle=1; Size=1; Stall=0; }
                                { Latency=1; Usage=1; }
        Control_ldjr ADDR       { Control.OP = 0xC ;
                                  Split.ADDRs = ADDR ; }
                                { JR <- ext22(ADDR); }
                                {}
                                { Cycle=1; Size=1; Stall=0; }
                                { Latency=1; Usage=1; }
        Control_ldjrpc          { Control.OP = 0xD ; }
                                { JR <- PC ; }
                                {}
                                { Cycle=1; Size=1; Stall=0; }
                                { Latency=1; Usage=1; }
        Control_HALT            { Control.OP = 0xE ; }
                                { HALT(); }
                                {}
                                { Cycle=1; Size=1; Stall=0; }
                                { Latency=1; Usage=1; }


// ////////////////////////////////////////////////////////////////////////

Field ALUf:
        ALU_NULL                                DEFINE_NULL_OP
        ALU_add     ALU_RA,ALU_RB,ALU_RW        { ALU.OP = 0x0 ;
                                                  ALU.RA = ALU_RA ;
                                                  ALU.RB = ALU_RB ;
                                                  ALU.RW = ALU_RW ; }
                                  { ALU_RW <-  ADDm(ALU_RA, ALU_RB) ; }
                                  {}
                                  { Cycle=1; Size=1; Stall=0; }
                                  { Latency=1; Usage=1; }
        ALU_addc    ALU_RA,CONST,ALU_RW         { ALU.OP = 0x1 ;
                                                  ALU.RA = ALU_RA ;
                                                  Split.CONSTs = CONST ;
                                                  ALU.RW = ALU_RW ; }
                          { ALU_RW <-  ADDm(ALU_RA, sx10(CONST)) ; }
```

```
                        {}
                        { Cycle=1; Size=1; Stall=0; }
                        { Latency=1; }
Control_call            { Control.OP = 0x1 ; }
                        { STACK[SP] <- PC + 1;
                          SP <- SP + 1; PC <- JR ; }
                        {}
                        { Cycle=1; Size=1; Stall=0; }
                        { Latency=1; Usage=1; }
Control_rtn             { Control.OP = 0x2 ; }
                        { SP <- SP - 1; PC <- STACK[SP]; }
                        {}
                        { Cycle=1; Size=1; Stall=0; }
                        { Latency=1; Usage=1; }
Control_jump            { Control.OP = 0x3 ; }
                        { PC <- JR ; }
                        {}
                        { Cycle=1; Size=1; Stall=0; }
                        { Latency=1; Usage=1; }
Control_jumpcz RI       { Control.OP = 0x4 ;
                          Control.RI = RI ; }
                        { if (RI == 0) { PC <- JR ; }; }
                        {}
                        { Cycle=1; Size=1; Stall=0; }
                        { Latency=1; Usage=1; }
Control_jumpcp RI       { Control.OP = 0x5 ;
                          Control.RI = RI ; }
                        { if (RI > 0) { PC <- JR ; }; }
                        {}
                        { Cycle=1; Size=1; Stall=0; }
                        { Latency=1; Usage=1; }
Control_rti             { Control.OP = 0x6 ; }
                        { SP <- SP - 1; PC <- STACK[SP]; }
                        {}
                        { Cycle=1; Size=1; Stall=0; }
                        { Latency=1; Usage=1; }
Control_trap TI         { Control.OP = 0x7 ;
                          Control.RI = TI ; }
                        { STACK[SP] <- PC + 1;
                          SP <- SP + 1;
                          PC <-  SUBm(ext6(TI), 0x00000002); }
                        {}
                        { Cycle=1; Size=1; Stall=0; }
                        { Latency=1; Usage=1; }
Control_rtt             { Control.OP = 0x8 ; }
                        { SP <- SP - 1; PC <- STACK[SP]; }
                        {}
                        { Cycle=1; Size=1; Stall=0; }
                        { Latency=1; Usage=1; }
Control_br OFFSET       { Control.OP = 0x9 ;
                          Split.OFFSETs = OFFSET ; }
                        { PC <-  ADDm(PC, sx16(OFFSET)) ; }
                        {}
                        { Cycle=1; Size=1; Stall=0; }
```

```
ProgramCounter PC               = 0x20          // program counter
Stack STACK(SP)                 = 0x10 , 0x20

// ---------------------------------------------------------------------------
// Correspondence between assembly mneumonics, bitfields, and actual
// instructions.

Section Instruction_Set

//                         RTL Descriptions
// NOTES:
//
// 1) There is the possibility that one instruction might have two
// different RTL descriptions based on some piece of state that is
// visible to the compiler. In this case we can use a CASE statement
// selecting over such state to describe which RTL description is
// appropriate for such an instruction. Note that conditional control
// flow instructions make use of this construct.
//
// 2) To describe the flow of control, we describe the values received
// by a specially declared register (declared in the storage section)
// usually called the PC. The value received by this register is the
// address from which the next instruction will be fetched.
//
// 3) NULL is a specially defined register which returns an arbitrary
// value when read and has no effect when written.
//
// here are the definitions of the operators

#define DEFINE_NULL_OP  {} { NULLOP(); } {} {} {}

#define ext3(x)      EXT(x,3,32)
#define ext6(x)      EXT(x,6,32)
#define ext10(x)     EXT(x,10,32)
#define ext22(x)     EXT(x,22,32)
#define sx16(x)      SEXT(x,16,32)
#define sx10(x)      SEXT(x,10,32)
#define ADDm(x,y)    ADD(x,y,32,"trn")
#define SUBm(x,y)    SUB(x,y,32,"trn")
#define MULm(x,y)    MUL(x,y,32,32,"trn")
#define DIVm(x,y)    DIV(x,y,32,32,32,"trn")
#define IORm(x,y)    OR(x,y,32)
#define ANDm(x,y)    AND(x,y,32)
#define NOTm(x)      NOT(x,32)
#define XORm(x,y)    XOR(x,y,32)
#define LSLm(x,y)    ASL(x,y,0,NULL,32)
#define LSRm(x,y)    ASR(x,y,0,NULL,32)
#define FADDm(x,y)   FADD(x,y,8,24,"sat")
#define FMULm(x,y)   FMUL(x,y,8,24,"sat")

Field Control:
        Control_NULL            DEFINE_NULL_OP
        Control_NOP             { Control.OP = 0x0 ; }
                                { NOP(); }
```

```
                            ALU_R   { DB.SRC = 0x4; ALU.RMEM = ALU_R; }
                                    {ALU[ALU_R]} {} {} {} |
                            MAC_R   { DB.SRC = 0x5; MAC.RMEM = MAC_R; }
                                    {MAC[MAC_R]} {} {} {} |
                            DATA    { DB.SRC = 0x6; Split.DATAs = DATA;}
                                    {DATA}  {} {} {} |
                            EIUCRt  { DB.SRC = 0x7; } {EIUCR} {} {} {} |
                            EIUARt  { DB.SRC = 0x8; } {EIUAR} {} {} {} |
                            EIUDRt  { DB.SRC = 0x9; } {EIUDR} {} {} {} |
                            CTRLt   { DB.SRC = 0xA; } {CTRL}  {} {} {} ;

Non_Terminal SINK:          AG1_R   { DB.SINK = 0x2; AG1.RMEM = AG1_R; }
                                    {AG1[AG1_R]} {} {} {} |
                            AG2_R   { DB.SINK = 0x3; AG2.RMEM = AG2_R; }
                                    {AG2[AG2_R]} {} {} {} |
                            ALU_R   { DB.SINK = 0x4; ALU.RMEM = ALU_R; }
                                    {ALU[ALU_R]} {} {} {} |
                            MAC_R   { DB.SINK = 0x5; MAC.RMEM = MAC_R; }
                                    {MAC[MAC_R]} {} {} {} |
                            EIUCRt  { DB.SINK = 0x6; } {EIUCR} {} {} {} |
                            EIUARt  { DB.SINK = 0x7; } {EIUAR} {} {} {} |
                            EIUDRt  { DB.SINK = 0x8; } {EIUDR} {} {} {} |
                            CTRLt   { DB.SINK = 0x9; } {CTRL}  {} {} {} |
                            IDLE    { DB.SINK = 0xC; } {NULL}  {} {} {} ;


Split.OFFSETs    AG2.OP+AG2.RW+AG2.RA+AG2.RB+AG2.RMEM;
Split.ADDRs      Control.RI+AG2.OP+AG2.RW+AG2.RA+AG2.RB
                 +AG2.RMEM;
Split.DATAs      AG2.OP+AG2.RW+AG2.RA+AG2.RB+AG2.RMEM
                 +AG1.OP+AG1.RW+AG1.RA+AG1.RB+AG1.RMEM;
Split.CONSTs     ALU.RB+ALU.RMEM;

// ---------------------------------------------------------------------------
// The number of registers for each unit and the topology of
// any of the register files and memories.

Section Storage

//                              = depth , width
Instruction Memory IM           = 0x100000 , 0x63
Memory DM1                      = 0x100000 , 0x20
Memory DM2                      = 0x100000 , 0x20
RegFile ALU                     = 0x20 , 0x20
RegFile MAC                     = 0x10 , 0x20
RegFile AG1                     = 0x8 , 0x20
RegFile AG2                     = 0x8 , 0x20
Register ACC                    = 0x20
Register SP                     = 0x4           // stack pointer
CRegister EIUCR                 = 0x20
CRegister EIUAR                 = 0x20
CRegister EIUDR                 = 0x20
CRegister CTRL                  = 0x20
CRegister JR                    = 0x20          // jump register
```

```
// This ends up in the lex file.
//       assembly        token   value
Token   "AG1.R"[0..7]   AG1_R   { [0..7]; };
Token   "AG2.R"[0..7]   AG2_R   { [0..7]; };
Token   "ALU.R"[0..31]  ALU_R   { [0..31]; };
Token   "MAC.R"[0..15]  MAC_R   { [0..15]; };
Token   EIUCR           EIUCRt  { };
Token   EIUAR           EIUARt  { };
Token   EIUDR           EIUDRt  { };
Token   CTRL            CTRLt   { };
Token   IDLE            IDLE    { };


// This ends up in the yacc file.
//            assembly        token   value
Non_Terminal RI:           AG1_R   {$$ = AG1_R; }
                                    { AG1[AG1_R] } {} {} {} |
                           AG2_R   {$$ = AG2_R|8; }
                                    { AG2[AG2_R] } {} {} {} |
                           ALU_R   {$$ = ALU_R|16; }
                                    { ALU[ALU_R] } {} {} {} |
                           MAC_R   {$$ = MAC_R|48; }
                                    { MAC[MAC_R] } {} {} {} ;
Non_Terminal TI:           INT     {$$ = INT; } { INT }   {} {} {} ;
Non_Terminal AG2_RA:       AG2_R   {$$ = AG2_R; } { AG2[AG2_R] } {} {} {} ;
Non_Terminal AG2_RB:       AG2_R   {$$ = AG2_R; } { AG2[AG2_R] } {} {} {} ;
Non_Terminal AG2_RMEM:     AG2_R   {$$ = AG2_R; } { AG2[AG2_R] } {} {} {} ;
Non_Terminal AG2_RW:       AG2_R   {$$ = AG2_R; } { AG2[AG2_R] } {} {} {} ;
Non_Terminal AG1_RA:       AG1_R   {$$ = AG1_R; } { AG1[AG1_R] } {} {} {} ;
Non_Terminal AG1_RB:       AG1_R   {$$ = AG1_R; } { AG1[AG1_R] } {} {} {} ;
Non_Terminal AG1_RMEM:     AG1_R   {$$ = AG1_R; } { AG1[AG1_R] } {} {} {} ;
Non_Terminal AG1_RW:       AG1_R   {$$ = AG1_R; } { AG1[AG1_R] } {} {} {} ;
Non_Terminal ALU_RA:       ALU_R   {$$ = ALU_R; } { ALU[ALU_R] } {} {} {} ;
Non_Terminal ALU_RB:       ALU_R   {$$ = ALU_R; } { ALU[ALU_R] } {} {} {} ;
Non_Terminal ALU_RMEM:     ALU_R   {$$ = ALU_R; } { ALU[ALU_R] } {} {} {} ;
Non_Terminal ALU_RW:       ALU_R   {$$ = ALU_R; } { ALU[ALU_R] } {} {} {} ;
Non_Terminal MAC_RA:       MAC_R   {$$ = MAC_R; } { MAC[MAC_R] } {} {} {} ;
Non_Terminal MAC_RB:       MAC_R   {$$ = MAC_R; } { MAC[MAC_R] } {} {} {} ;
Non_Terminal MAC_RMEM:     MAC_R   {$$ = MAC_R; } { MAC[MAC_R] } {} {} {} ;
Non_Terminal MAC_RW:       MAC_R   {$$ = MAC_R; } { MAC[MAC_R] } {} {} {} ;

Non_Terminal OFFSET:       INT     {$$ = INT; } { INT }   {} {} {} |
                           NAME    {$$ = NAME - CURRENT; }
                                    { NAME - PC } {} {} {} ;

Non_Terminal ADDR:         INT     {$$ = INT; } { INT }   {} {} {} |
                           NAME    {$$ = NAME; } { NAME } {} {} {} ;

Non_Terminal DATA:         INT     {$$ = INT; } { INT } {} {} {} ;
Non_Terminal CONST:        INT     {$$ = INT; } { INT } {} {} {} ;
Non_Terminal SRC:          AG1_R   { DB.SRC = 0x2; AG1.RMEM = AG1_R; }
                                        {AG1[AG1_R]} {} {} {} |
                           AG2_R   { DB.SRC = 0x3; AG2.RMEM = AG2_R; }
                                        {AG2[AG2_R]} {} {} {} |
```

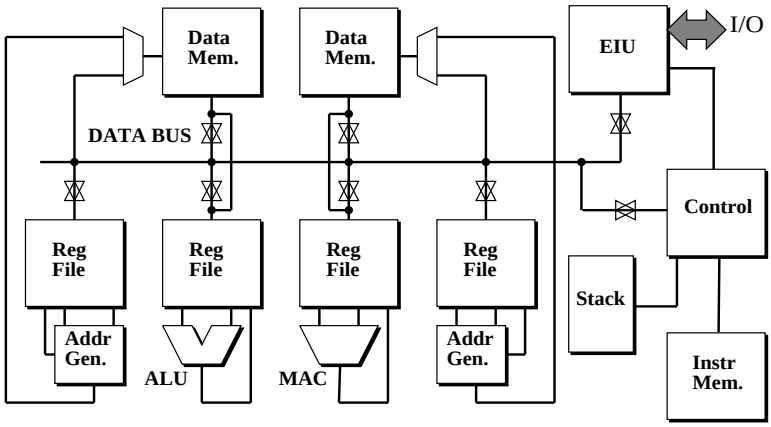

Figure 8: **The Sample SPAM VLIW processor.**

C Example Descriptions

C.1 The SPAM Sample VLIW

```
// Conventions:

// | = OR
// & = AND
// [x..y] = range from x to y
// , = used between required fields
// @ = variable declaration follows

// \ = ignore special symbol


// ---------------------------------------------------------------------------
// Number of fields in each VLIW word
// Size and format of each

Section Format

Control = OP[4], RI[6];
AG2     = OP[4], RW[3], RA[3], RB[3], RMEM[3];
AG1     = OP[4], RW[3], RA[3], RB[3], RMEM[3];
ALU     = OP[4], RW[5], RA[5], RB[5], RMEM[5];
MAC     = OP[3], RW[4], RA[4], RB[4], RMEM[4];
DB      = SRC[4], SINK[4];
ALUOP   = OP[2];
MACOP   = OP[2];
DM1A    = OP[1];
DM2A    = OP[1];


// ---------------------------------------------------------------------------

Section Global_Definitions
```

```
                                  <Regex_Constant> | <Regex_Range>
                                  <Regular_Expression> <Regular_Expression>

<Constraint_Unary_Operator>     := '~' | <Time_Shift_Operator>

<Constraint_Binary_Operator>    := '&' | '|'

<Time_Shift_Operator>           := <index_operator>

<Widlcard>                      := '?' | '+' | '*'

<Variable_Match>                := '@' '[' DIGIT ']'

<Regex_Range>                   := RGX_RNG

<Regex_Constant>                := NAME | <simple_char> | DIGIT |
                                  <escaped_char>

<simple_char>                   := SPACE | '$' | '~' | '`' | ''' | '{' |
                                  '!' | '%' | '-' | '}' | '>' | '<' |
                                  ',' | '.' | ':' | ';' | '/'

<escaped_char>                  := '\@' | '\^' | '\&' | '\*' | '\+' |
                                  '\?' | '\[' | '\]' | '\|'



<Optional>      := 'Section' 'Optional'
```

```
<RTL_Unary_Operator>    := '~' | '!' | '-'

<RTL_Binary_Operator>   := '+' | '-' | '*' | '/' | '%' | '>>' | '<<' |
                           '==' | '<' | '>' | '<=' | '>='

<RTL_Function_Call>     := NAME '(' <RTL_Parameter_List> ')' |
                           NAME '(' ')'

<RTL_Parameter_List>    := <RTL_Parameter> |
                           <RTL_Parameter> ',' <RTL_Parameter_List>

<RTL_Parameter>         := <RTL_Expression> | STRING

<CT_Clause>             := <CT_Assign_List>

<CT_Assign_List>        := <CT_Assign> |
                           <CT_Assign> <CT_Assign_List> |

<CT_Assign>             := NAME '=' <CT_Expression> ';'

<CT_Expression>         := <constant> | <Token_Name> |
                           <Non_Terminal_Name> | <Field_Name> |
                           <Storage_Reference> |
                           <CT_Unary_Operator> <CT_Expression> |
                      <CT_Expression> <CT_Binary_Operator> <CT_Expression> |
                           <CT_Function_Call>

<CT_Unary_Operator>     := '-'

<CT_Binary_Operator>    := '+' | '-' | '*' | '/' | '%' | '==' | '<' |
                           '>' | '<=' | '>='

<CT_Function_Call>      := NAME '(' <CT_Parameter_List> ')'
                           NAME '(' ')'

<CT_Parameter_List>     := <CT_Expression> |
                           <CT_Expression> ',' <CT_Parameter_List>



<Constraints>   := 'Section' 'Constraints' <Constraint_List>

<Constraint_List>       := <Constraint> |
                           <Constraint> <Constraint_List> |

<Constraint>    := <Constraint_Expression>

<Constraint_Expression> := '(' <Regular_Expression> ')' |
               <Constraint_Unary_Operator> <Constraint_Expression> |
               '(' <Constraint_Expression> <Constraint_Binary_Operator>
                   <Constraint_Expression> ')'

<Regular_Expression>            := <Widlcard> | <Variable_Match> |
```

```
<RTL_Statement_List>        := <RTL_Statement> |
                               <RTL_Statement> <RTL_Statement_List> |

<RTL_Statement>             := <RTL_Declaration> | <RTL_Assignment> |
                               <RTL_If_Clause> | <RTL_For_Clause> |
                               <RTL_While_Clause> | <RTL_Switch_Clause>
                               NAME ';' | <RTL_Function_Call> ';'

<RTL_Declaration>       := 'int' <number> NAME ';'

<RTL_Assignment>        := <RTL_Assign_Left> '<-' <RTL_Expression> ';'

<RTL_If_Clause>         := 'if' '(' <RTL_Expression> ')'
                                   '{' <RTL_Statement_List> '}' ';' |
                           'if' '(' <RTL_Expression> ')'
                                   '{' <RTL_Statement_List> '}'
                                   'else' '{' <RTL_Statement_List> '}' ';'

<RTL_For_Clause>        := 'for' '(' <RTL_Statement> <RTL_Expression>
                       <RTL_Statement> ')' '{' <RTL_Statement_List> '}' ';'

<RTL_While_Clause>      := 'while' '(' <RTL_Expression> ')'
                           '{' <RTL_Statement_List> '}' ';'

<RTL_Switch_Clause>     := 'switch' '(' <RTL_Expression> ')' '{'
                       <Switch_Case_List> <Switch_Optional_Default> '}' ';'

<Switch_Case_List>      := <Switch_Case> |
                           <Switch_Case> <Switch_Case_List>

<Switch_Case>           := 'case' <number> ':' '{'
                                <RTL_Statement_List> '}' ';'

<Switch_Optional_Default>   := 'default' ':' '{' <RTL_Statement_List> '}' |

<RTL_Assign_Left>       := <Storage_Reference> | <Token_Name> |
                           <Non_Terminal_Name> | <Tmp_Name> | 'NULL'

<Token_Name>            := NAME

<Non_Terminal_Name>     := NAME

<Tmp_Name>              := NAME

<RTL_Expression>        := <constant> | <Storage_Reference> | <Token_Name> |
                           <Non_Terminal_Name> | <Tmp_Name> |
                           <RTL_System_Flag> |
                           <RTL_Unary_Operator> <RTL_Expression> |
                    <RTL_Expression> <RTL_Binary_Operator> <RTL_Expression> |
                    <RTL_Function_Call> | '(' <RTL_Expression> ')'

<RTL_System_Flag>       := NAME | <RTL_Function_Call>
```

```
<Instruction>   := 'Section' 'Instruction_Set' <Inst_Field_List>

<Inst_Field_List>       := <Inst_Field> |
                           <Inst_Field> <Inst_Field_List>

<Inst_Field>    := 'Field' NAME ':' <Operation_Definition_List>

<Operation_Definition_List> := <Operation_Definition> |
                        <Operation_Definition> <Operation_Definition_List>

<Operation_Definition>  := <Operation_Syntax> <Operation_Bitfield_Assign>
                           <Operation_Action> <Operation_Side_Effects>
                           <Operation_Costs> <Operation_Timing>

<Operation_Syntax>      := NAME <Operation_Parameter_List> |
                           NAME

<Operation_Parameter_List>  := NAME |
                               NAME ',' <Operation_Parameter_List>

<Operation_Bitfield_Assign> := '{' <Bitfield_Assign_List> '}'

<Bitfield_Assign_List>  := <Bitfield_Assign> |
                           <Bitfield_Assign> <Bitfield_Assign_List> |

<Bitfield_Assign>       := <Bitfield_Assign_Statement> |
                           <Additional_Word_Clause>

<Bitfield_Assign_Statement> := <Bitfield_Assign_Left> '='
                               <Bitfield_Assign_Expression> ';'

<Bitfield_Assign_Left>  := <Subfield_Name> | 'Split' '.' NAME

<Bitfield_Assign_Expression>    := <number>                              |
                                   NAME                                  |
                                   'CURRENT'                             |
                                   Subfield_Name                         |
                                   '(' <Bitfield_Assign_Expression> ')'  |
                                   '~' <Bitfield_Assign_Expression>      |
                        <Bitfield_Assign_Expression> <BA_Binary_Operator>
                                    <Bitfield_Assign_Expression>

<BA_Binary_Operator>    := '&' | '|' | '^' | '+' | '*' | '>>' | '<<'

<Additional_Word_Clause>    := 'Additional' '(' <number> ','
                               <Bitfield_Assign_Statement> ')' ';'

<Operation_Action>          := '{' <RTL_Statement_List> '}'

<Operation_Side_Effects>    := '{' <RTL_Statement_List> '}'

<Operation_Costs>           := '{' <CT_Clause> '}'

<Operation_Timing>          := '{' <CT_Clause> '}'
```

```
<Storage>       := 'Section' 'Storage' <Storage_Definition_List>

<Storage_Definition_List>   := <Storage_Definition> |
                  <Storage_Definition> <Storage_Definition_List>

<Storage_Definition>    := <IMEM_Definition>    |
                           <MEM_Definition>     |
                           <RegFile_Definition> |
                           <Reg_Definition>     |
                           <CReg_Definition>    |
                           <MMIO_Definition>    |
                           <PC_Definition>      |
                           <Stack_Definition>   |
                           <Alias_Definition>

<IMEM_Definition>       := 'Instruction' 'Memory' NAME '=' <Adressed_Size>

<MEM_Definition>        := 'Memory' NAME '=' <Adressed_Size>

<RegFile_Definition>    := 'RegFile' NAME '=' <Adressed_Size>

<Reg_Definition>        := 'Register' NAME '=' <Register_Size>

<CReg_Definition>       := 'CRegister' NAME '=' <Register_Size>

<MMIO_Definition>       := 'MMIO' NAME '=' <Adressed_Size>

<PC_Definition>         := 'ProgramCounter' NAME '=' <Register_Size>

<Stack_Definition>      := 'Stack' NAME '(' NAME ')' '=' <Adressed_Size>

<Alias_Definition>      := 'Alias' NAME <Storage_Reference_List> ';'
                           <Storage_Size>

<Adressed_Size>         := <number> ',' <number>

<Register_Size>         := <number>

<Storage_Size>          := <Adressed_Size> | <Register_Size>

<Storage_Reference_List>    := <Storage_Reference> |
                         <Storage_Reference> ',' <Storage_Reference_List>

<Storage_Reference>     := NAME                             |
                           NAME <IR_operator>               |
                           NAME <IR_operator> <IR_operator>

<IR_operator>           := <index_operator> | <range_operator>
```

```
<Token_Range_Operator>  := '[' <number> '..' <number> ']'

<Token_Return>      := '{' '}' | '{' <number> ';' '}'
                       '{' <Token_Range_Operator> ';' '}'


<Non_Terminal_Definition>   := 'Non_Terminal' NAME ':'
                               <Non_Terminal_Option_List> ';'

<Non_Terminal_Option_List>  := <Non_Terminal_Option> |
                    <Non_Terminal_Option> '|' <Non_Terminal_Option_List>

<Non_Terminal_Option>   := <NT_Option_Syntax_List> <NT_Option_Return>
                    <NT_Option_Action> <NT_Option_Side_Effect>
                    <NT_Option_Cost_Mod> <NT_Option_Timing_Mod>

<NT_Option_Syntax_List> := <NT_Option_Syntax_Item> |
                           <NT_Option_Syntax_Item> <NT_Option_Syntax_List>

<NT_Option_Syntax_Item> := NAME | CHAR

<NT_Option_Return>  := '{' <NT_Option_Return_Assignment_List> '}'

<NT_Option_Return_Assign_List>  := <NT_Option_Return_Assign_Item> |
                                   <NT_Option_Return_Assign_Item>
                                   <NT_Option_Return_Assign_List>

<NT_Option_Return_Assign_Item>  := <NTOR_Assign_Left> '='
                                   <Bitfield_Assign_Expression> ';'

<NTOR_Assign_Left>              := <Bitfield_Assign_Left> |
                                   '$$'

<NT_Option_Action>      := '{' <RTL_Partial> '}'

<NT_Option_Side_Effect> := '{' <RTL_Partial> '}'

<NT_Option_Cost_Mod>    := '{' <CT_Clause> '}'

<NT_Option_Timing_Mod>  := '{' <CT_Clause> '}'

<RTL_Partial>       := <RTL_Statement_List> | <RTL_Expression>

<Split_Function_Definition> := 'Split' '.' NAME
                               <Split_Function_Subfield_List> ';'

<Split_Function_Subfield_List>  := <Subfield_Name> |
                                   <Subfield_Name> '+'
                                   <Split_Function_Subfield_List>

<Subfield_Name> := NAME '.' NAME
```

B BNF Syntax For ISDL

```
NAME    := [a-zA-z][a-zA-Z0-9_]+
INT     := -?[0-9]+
HEX     := 0x[0-9A-F]+ | 0x[0-9a-f]+
FLOAT   := -?[0-9]+\.[0-9]+([eE]-?[0-9]+)?
STRING  := "[^"]*" | `[^`]*`
CHAR    := '.'
DIGIT   := [0-9]
SPACE   := [ \t\n]+
RGX_RNG := \[[^\]]*(\\\])?[^\]]*\]


<number>            := INT | HEX
<constant>          := INT | HEX | FLOAT
<range_operator>    := '[' <number> '-' <number> ']'
<index_operator>    := '[' <number> ']'




<ISDL_Description>  := <Format> <Definitions> <Storage> <Instruction>
                       <Constraints> <Optional>



<Format>            := 'Section' 'Format' <Format_Field_List>

<Format_Field_List> := <Format_Field> |
                       <Format_Field> <Format_Field_List>

<Format_Field>      := NAME '=' <Format_Subfield_List> ';'

<Format_Subfield_List>  := <Format_Subfield> |
                           <Format_Subfield> ',' <Format_Subfield_List>

<Format_Subfield>   := NAME <index_operator>




<Definitions>       := 'Section' 'Global_Definitions' <Definitions_List>

<Definitions_List>  := <Single_Definition> |
                       <Single_Definition> <Definitions_List>

<Single_Definition> := <Token_Definition>           |
                       <Non_Terminal_Definition>    |
                       <Split_Function_Definition>

<Token_Definition>  := 'Token' <Token_Syntax> NAME <Token_Return> ';'

<Token_Syntax>      := NAME | NAME <Token_Range_Operator> |
                       STRING | STRING <Token_Range_Operator>
```

- **Variable Length Instructions**: Some architectures have instructions that may be of different lengths (depending on the action that the instruction is taking). The instruction word of such an architecture is called a Variable Length Instruction.
- **Variable Match**: An operation match containing a variable binding wild-card. The first time the given variable binding occurs it will match zero or more characters of any type and store these in a variable. Any subsequent times it occurs, it will only match the string that was stored in the variable after the first match.
- **Visible State**: The collective state in a processor that can be explicitly inspected and/or modified by the use of a sequence of processor instructions. Thus, all registers that can be directly written to or read from with operations from the instruction set, are part of the visible state. All registers that cannot (such as temporary registers in certain functional units and the registers making up a pipeline) are not considered part of the visible state.
- **Wild-cards**: Special characters in regular expressions that can match more than a single character and may match one or more copies of such characters. For example "**?**" will match any single character while "*****" will match zero or more copies of any single character.
- **Yacc**: Yet Another Compiler Compiler[4]. A parser generator which takes a description of a context free grammar and generates a parser to parse it. The generated parser in combination with a lexical analyzer can be used to process inputs in a given language (hence the name compiler compiler). ISDL is processed using a parser generated by Yacc. The generated assemblers also use parsers generated by Yacc. Yacc generates SR parsers with single token lookahead and this must be taken into account when generating grammars for them.
- **Zero Overhead Looping**: A methodology of creating efficient software loops by performing all control overhead associated with the loop (such as decrementing a counter variable, checking the variable for zero or some other value, and performing flow control) in hardware. Since data dominated applications often make heavy use of small data manipulation loops, most DSP processors provide one or more forms of zero overhead looping.

- **Super-Scalar**: A way of allowing parallelism in the implementation of a processor without exposing this parallelism to the instruction set. Thus, super-scalar architectures typically have instruction sets which are reminiscent of unifunctional processors, but have implementations that contain multiple identical functional units and issue instructions to these units in parallel whenever precedence constraints in the software allow it. In effect, super-scalar architectures do some code-scheduling in hardware while VLIW architectures with multiple identical functional units do all their scheduling in software.

- **Time-Shifted Constraint**: A constraint that expresses a restriction between instructions issued at different times.

- **Timing**: A model describing when the effects of a given operation are visible to other operations. Usually expressed in terms of one or more numerical values.

- **Token**: A token is an ISDL abstraction which groups together one or more lexically related entities (such as the names of registers in a register file).

- **Token Assembly**: The part of a token definition that describes the assembly syntax of the lexically related entities grouped under a token.

- **Token Return Value**: Since tokens group together one or more lexical entities, it is necessary to provide a mechanism that identifies which of the lexical entities was present for a particular instantiation of a token. This is provided by the return value which is a numerical value unique to each of the lexical entities grouped under a token. The return value is often used to generate a binary image for the entity represented by the token, so the encodings of the return values should be chosen to make this process easier.

- **Trap**: An exception generated by the use of an operation specifically designed for this purpose. Just as in the case of normal exceptions, control is diverted to a fixed address where a trap handler attempts to service the exception. Traps may be, and often are, accompanied by changes in the mode of the processor (such as a change from user mode to privileged mode). They are often used to trigger execution of kernel functions in privileged mode, or to inform a piece of privileged code (usually the kernel) of a required service.

- **Trap Handler**: A small piece of code that is executed when a specific trap event occurs, and which attempts to service the trap.

- **Truncation Arithmetic**: A method of handling overflow (underflow) in arithmetic operations. If the result overflows (underflows), the appropriate flag is set and the result is set to as many bits as the destination can hold (i.e. the top few bits are truncated).

- **Unifunctional**: An architecture that can only perform one operation at a time. This does not necessarily mean that it only has one functional unit. It *does* mean that only one functional unit is active at any given time (ignoring program counter increment operations). Unifunctional architectures have no instruction level parallelism.

- **Usage(Timing)**: A timing parameter associated with each operation that declares when the "functional unit" executing the operation may be used again. A usage of 2 means that the "functional unit" (i.e. the field containing this operation) may not be used in the next instruction, and therefore a `NOOP` should be selected from the corresponding field in the next instruction. If any other operation is selected instead, the hardware will stall the next instruction until the functional unit becomes available once again.

- **VLIW**: Very Long Instruction Word. An architecture with instruction level parallelism, capable of performing multiple operations on multiple functional units at the same time. Each functional unit is typically independent of the others and has its own dedicated bitfields in the instruction word. This typically results in very large word lengths, hence the name.

increments the stack pointer first and then writes a value into the location pointed to by the new value of the stack pointer. It is an error to pop a value from an empty stack or to push a value onto a full stack. Typically only the value pointed to by the stack pointer (called the top of the stack) can be accessed even though a stack is a form of addressed storage.

- **Stack Pointer**: The special purpose register that provides the address for a stack.
- **Stall Cost**: The maximum additional number of cycles taken up by stalls, when the next instruction tries to use the results of the current operation. Note that this cost is derated linearly as the distance between the current operation and the instruction which attempts to use the results increases. In other words, if the stall cost is 3, and the instruction using the results of the current operation is the next instruction (distance 1), the actual number of stall cycles is 3. If the distance increases to two (i.e. the instruction using the results is the instruction after the next one) the actual number of stall cycles is 2, and so on until the stall cost falls to 0. Stall cost is one of the ISDL predefined costs.
- **State**: The collective amount of visible storage in a processor. This is equivalent to the storage of the processor, with the distinction that storage usually refers to the actual storage elements available while state usually refers to the values contained in these storage elements. A processor can be modeled in terms of its state and the operations that modify this state.
- **Storage**: The collective amount of storage elements present in a processor and visible to the programmer. This is equivalent to the state of the processor, with the distinction that storage usually refers to the actual storage elements available while state usually refers to the values contained in these storage elements.
- **Storage Alias**: An alternative name for a subset of the state of a processor. It usually groups together subsets of the storage already defined and treats them as new storage units, with the exception that changes in the subset affect the value read from the newly defined state (the aliased state) and vice versa.
- **Storage Depth**: Addressed storage can be considered as a collection of registers (special purpose or otherwise). The number of such registers that make up an addressed storage unit is referred to as the *depth* of the storage unit. The *depth* of a single register (a single storage element) is 1.
- **Storage Reference**: An expression identifying a subset of the storage available in a processor. It consists of a name, possibly an address or range of addresses (using a range operator), and/or a subset of bits within the named storage unit. The name can be the identifier associated either with a properly defined storage unit or the identifier of a storage alias.
- **Storage Section**: The section of an ISDL description that describes the storage units available to the programmer. It consists of a list of storage unit definitions and, possibly, a number of storage aliases.
- **Storage Type**: The type of a storage unit. ISDL currently defines the following types of storage units: memories (including instruction memories), register files, stacks, registers, control registers, a program counter and memory-mapped I/O locations.
- **Storage Width**: The width in bits of a storage element. For a register (control or otherwise) this is the total number of bits in the register. For an addressed storage unit, this refers to the width of each element in the storage unit (which should be the same for all elements in the storage unit).
- **Structural Language**: A language that describes a target architecture by giving the structure of the architecture (i.e. by describing the functional units and storage elements in the architecture and how these are connected together). Structural languages contain much information which is not relevant to code generation or instruction level simulation, but can be used to generate both the tools as well as an implementation of the architecture.

that are capable of executing each instruction in one clock cycle and have very fast clock speeds. This gives them a high throughput but results in larger code size for the same input source code (note that this may make instruction caches less efficient and thus reduce effective throughput).

- **RTL Functions**: A provision in the version of RTL that ISDL supports, to call functions to perform specialized data manipulation (such as floating point operations etc.). ISDL predefines a set of RTL functions; more may be defined and used if the tools that process ISDL are updated to make use of them.

- **Range Operator**: An operator that denotes a range of numbers or characters by listing the beginning value and the ending value of the range.

- **Register File**: An addressed storage unit, typically equipped with multiple ports, through which a functional unit may access multiple registers. The registers are typically of the same width. Register files are typically used to provide the source and destination operands to operations in load/store architectures. They are, more often than not, closely coupled to one or more functional units and can therefore be accessed in a single clock cycle.

- **Register Transfer Language**: RTL. A type of language well suited to describing the behavior of data paths. The main type of statement is an assignment which assigns a new value to a piece of state. The language also contains a number of control features that allow it to specify the behavior of control circuitry as well. ISDL, however, has its own concept of control so it only makes use of a restricted subset of the language to describe the behavior of the data paths on a per-operation basis.

- **Regular Expressions**: An expression syntax specifically designed to describe regular grammars. It makes heavy use of wild-card characters. ISDL regular expressions are augmented with variable matches which require the usage of a stack.

- **Regular Grammar**: A grammar is a set of input strings. A regular grammar is one that can be identified by an FSM.

- **Resource Conflict**: An attempt by two different operations to use the same data path resource (such as a register, data bus, or port on a register file) for different purposes. If an architecture's instruction encoding exposes resource conflicts to the instruction set, then a number of restrictions must be placed on the operations to avoid such conflicts. Operations that result in a resource conflict can never be executed successfully by the hardware.

- **Retargetable Compiler**: A compiler which is capable of emitting output code for new architectures when provided with a description of these architectures.

- **Retargetable Simulator**: A simulator which has the ability to obtain a description of a new architecture and assume the task of simulating that particular architecture.

- **Saturation Arithmetic**: A method of handling overflow (underflow) in arithmetic operations. If the result overflows (underflows), the appropriate flag is set and the largest (smallest) number that can be represented by the architecture is produced as the result.

- **Split Function**: A function that can take a large binary constant and split it into a number of bitfields. These functions are automatically generated from split function definitions in the Global Definitions Section.

- **Stack**: A type of storage unit that behaves like a stack of plates: i.e. multiple values can be stored and the last value stored will be the first one read. It is an addressed storage unit which uses a special purpose register (the stack pointer) to provide the address, and two special purpose operations to read and write the contents. *Pop* reads the contents of the location pointed to by the stack pointer and decrements the stack pointer automatically. *Push*

- **Operation Assembly Definition**: The part of an operation definition that provides the assembly syntax for the operation. It is also used to give a name to the operation. It is written in terms of an operation name and a list of parameters which may be tokens or non-terminals.
- **Operation Bitfield Assignment**: The part of an operation definition that contains the description of how to perform all the bitfield assignments for the binary image of the operation. It is written in a restricted expression form based on assignments.
- **Operation Costs**: The part of an operation definition that declares the costs associated with an operation. All costs in ISDL are numerical.
- **Operation Definition**: A set of ISDL clauses which fully defines a single operation within an instruction field.
- **Operation Match**: A regular expression that returns *true* if an operation (or one of a group of operations) is present in an instruction.
- **Operation RTL Action**: A description of the desired effects of the operation on processor state, written in an RTL-type language. If the operation makes use of non-terminals, the RTL action definition of the operation may refer to the RTL action definition of the non-terminals.
- **Operation RTL Side Effect**: A description of the side effects of an operation on visible state, written in an RTL-type language. If the operation makes use of non-terminals, the side effects definition of the operation may refer to the side effects definition of the non-terminals.
- **Operation Timing**: The part of an operation definition that declares the timing parameters associated with an operation. All timing parameters in ISDL are numerical. Timing parameters are mainly concerned with when the effects of an operation become visible in the state of the machine.
- **Optional Architectural Information Section**: A section of an ISDL description that provides information on the target architecture that might not be necessary for useful design tools to be generated, but may result in better tools. Information on the cache system is a good example - this information is not necessary to generate a code-generator or a simulator but both might benefit from the extra information.
- **Orthogonal Operations**: Operations that can be performed in parallel and do not affect each other in any way. In particular, the presence of one operation cannot preclude the presence of the other and the form of one operation cannot restrict the form of the other. In VLIW architectures, all operations would be orthogonal if there were no hardware restrictions. In practice this is rarely the case.
- **Partitioning (Hardware/Software)**: The process of dividing a task into two sub-parts and assigning one part to a custom hardware (i.e. ASIC) implementation and the remainder to a software implementation. The main goal of the process is to select the partition between the two parts so that cost and performance are optimized.
- **Program Counter**: A special purpose register present in every architecture, which points to the current (or next in some cases) instruction in the instruction stream. Typically the contents of this register cannot be explicitly transferred to another location (i.e. read). Furthermore, changing the value of this register diverts control to a new point in the instruction stream and the value automatically gets updated on every instruction fetch. In this respect, it behaves like a control register which can only be written using control flow operations and cannot be read explicitly.
- **RISC**: Reduced Instruction Set Computer. A processor with a simple architecture and a small instruction set. Typically such processors have very few addressing modes (most such processors are load/store architectures). They often have simple pipeline implementations

normal memory (sometimes using exactly the same operations). However, writes to memory-mapped I/O locations change the state of I/O peripherals and ports, and reads inspect the values and state on I/O peripherals and ports.

- **Micro-controller**: A processor designed mainly to control other peripherals. Such processors typically have small instruction words and narrow data paths (usually 8-bit instructions and 8-bit data paths) but have a characteristically disproportionate amount of I/O capability (such as three 8-bit bi-directional ports, a serial port and an external memory bus on an 8-bit micro-controller). They appear mostly in control-dominated applications.

- **Multiple Operation Definitions**: Operation definitions within the same field that share a common operation name but different numbers and/or types of arguments.

- **Non-Terminal**: An abstraction that groups syntactically unrelated entities into a logical group. Conceptually, a non-terminal consists of a number of alternatives, any of which may be replace the non-terminal in an actual instance of an operation. These alternatives are called options.

- **Non-Terminal Costs Clause**: Different options in a single non-terminal may result in different costs in an operation that uses this non-terminal. Non-terminals contain a set of cost expressions that allow operations to account for this by providing each option with its own set of cost expressions. This set of expressions is called a costs clause.

- **Non-Terminal Option**: One of the many possible alternatives that a non-terminal groups together into an abstraction. Each option consists of a syntax definition, a return value, an RTL action, an RTL side effect, a costs clause, and a timing clause.

- **Non-Terminal RTL Action**: This is the part of a non-terminal option definition that describes the action of the particular option (usually simply a storage reference). Each option in the non-terminal definition has its own RTL action clause.

- **Non-Terminal RTL Side Effect**: This is the part of a non-terminal option definition that describes the side effects of the particular option (usually simply a storage reference). Each option in the non-terminal definition has its own side effects clause.

- **Non-Terminal Return Value**: Since a non-terminal is a conceptual abstraction of a number of different options, there must be a mechanism of differentiating between options once a non-terminal is instantiated. This is done by providing a return value which is different for each option in the non-terminal. This return value can then be used to perform the bitfield assignment for the non-terminal so it is very common to encode the return values in such a way that they are identical to the binary image of the non-terminal.

- **Non-Terminal Timing Clause**: Different options in a single non-terminal may result in different timing in an operation that uses this non-terminal. Non-terminals contain a set of timing expressions that allow operations to account for this by providing each option with its own set of timing expressions. This set of expressions is called a timing clause.

- **Op-code**: Operation Code. A bitfield whose value typically uniquely identifies the *type* of operation to be performed by a processor or functional unit.

- **Operation**: The smallest unit of data manipulation that can be independently performed by the hardware. In VLIW architectures this is by definition a sub-part of the instruction since each VLIW instruction can be considered to be performing a number of independent data manipulations on each of its functional units. In other words, a VLIW instruction consists of multiple operations. In unifunctional architectures, the operation *is* by definition the instruction.

- **Instruction Level Simulator**: A simulator for a target architecture that simulates effects visible in the instruction set but no lower than that. For example, the simulator may give cycle-accurate simulation for the target architecture but not have an explicit model of the pipeline and thus make it impossible to inspect the state of the pipeline at any given time (the pipeline is not generally visible in the instruction set).

- **Instruction Set Section**: A section of an ISDL description that contains the definitions for all the operations available in the instruction set of the target architecture.

- **Interrupt**: A hardware signal that triggers a control flow operation in the processor. Control is diverted away from normal program execution and to a fixed address which may or may not depend on the signal that triggered the control flow operation. A small program called an interrupt handler resides at this address. This program attempts to deal with whatever condition asserted the hardware signal in the first place. Typically interrupts are used by peripherals to signal to the processor that they require attention. Interrupts may also cause mode changes in the processor (such as a switch from normal user mode to privileged mode).

- **Interrupt Handler**: A small piece of code that is executed when a specific interrupt event occurs and which attempts to service the interrupt.

- **Latency**: The number of instructions that need to be fetched before the effects of the current operation become visible.

- **Lex**: A lexical analyzer generator (a tool that receives as input a file containing regular expressions describing the lexical entities of a language and generates as output code that will implement a lexical analyzer)[4]. The lexical analyzer that is used to process ISDL descriptions was generated by Lex. The generated assemblers also use lexical analyzers generated by Lex to process their input files.

- **Lexical Analyzer**: A program that divides an input stream of characters into lexical entities. Together with a parser it can be used to parse input files written in various languages. ISDL descriptions are parsed using a lexical analyzer generated by Lex. The generated assemblers also use lexical analyzers generated by Lex to process their input files.

- **Load/Store Architectures**: A class of architectures in which all operations except loads and stores take their inputs from and write their results to registers.

- **Loop Counter**: A special purpose register in the data path used to keep count of how many iterations a zero overhead loop has performed (or has remaining).

- **Loop Destination Address**: The address of the first instruction in a zero overhead loop (where the loop will branch back to for each iteration until it terminates).

- **Loop Termination Address**: The address of the last instruction in a zero overhead loop (after which the test is performed to see if the loop terminated or if another iteration should be executed).

- **Macro Definitions**: Definitions of text to be expanded into longer and more complicated pieces of text by a preprocessor, before the description is actually processed by the tools. It allows common text patterns to be given shorter names in order to shrink the size of a description.

- **Memory Management Hardware**: Hardware that handles address translation schemes (such as the ones needed for virtual memory).

- **Memory-Mapped I/O**: A group of control registers that appears as memory to the Instruction Set but has the function of communicating with I/O peripherals. Typically, a processor can read from and write to these locations in the same way that it would read from or write to

- **Exception Handler**: A small piece of code that is executed when a specific exception condition occurs, and which attempts to recover from the exception condition.
- **Field Definition**: A list of operation definitions that are mutually exclusive, grouped together in the Instruction Set Section of an ISDL description. Typically corresponds to the operations that can be performed on a single functional unit in a VLIW processor.
- **Fixed Length Instructions**: The instruction word of a processor is called a fixed length instruction if all instructions available in the instruction set of the processor have the same word length. This does *not* include additional words that may be needed to provide large constants.
- **Flag**: Usually a single-bit piece of state (often provided as a bit in a control register) that denotes a certain condition (e.g. overflow) has occurred, or that denotes a current mode (e.g. interrupts disabled). Flags can be, and sometimes are, longer than a single bit.
- **Format Section**: The section of an ISDL description that describes the structure of the instruction word - i.e. the division into logical units called bitfields.
- **Global Definitions Section**: The section of an ISDL description that defines abstractions (such as tokens, non-terminals, and split functions) that are used in later sections of the description.
- **Hardware/Software Co-design**: A design methodology targeted mainly towards embedded systems that have both a hardware and a software component and attempt to integrate both components on a single chip. Such systems have the property that decisions concerning the hardware component drastically affect the software component and vice-versa. Because of this, the most effective way to design such systems is to provide a common framework for designing and evaluating both components together.
- **Hint Driven Branch Prediction**: A method of branch prediction where the guess as to the outcome of the branch is provided explicitly by the branch instruction used. Thus, the branch prediction mechanism is completely under the programmer's control. It can be achieved by providing duplicate branch instructions with exactly the same action except for the fact that one causes the guess to be that control will be diverted to the new location and the other causes the guess to be that control will remain sequential.
- **ISDL**: Instruction Set Description Language. A behavioral machine description language specifically designed to support a wide variety of architectures (including VLIW) and support the automatic generation of design environment tools. It closely models the structure of the Programmer's Manual.
- **Instruction**: The smallest logical unit that can be fetched by a processor. For VLIW architectures this might correspond to a number of operations, all grouped together into a single unit. For Super-Scalar architectures this may be a small part of what can be decoded and dispatched by the hardware.
- **Instruction Field**: A set of mutually exclusive but related operations in the instruction set of a target architecture. These typically (but not always) correspond to all the operations that a single functional unit in a VLIW architecture can perform.
- **Instruction Level Parallelism**: The ability to execute more than a single operation at any given time when this ability is visible in the instruction set of the target architecture. This, for example, would not encompass the parallelism in a super-scalar architecture, where the instruction set looks the same as that of a unifunctional architecture but operations *are* actually performed in parallel on multiple functional units.

- **Control Register**: A register which when written to may have side effects (such as resetting interrupt modes or writing values to an output port), and when read from is not guaranteed to return the last value written to it. Such registers are usually used to control various modes of the processor and peripherals, and to inspect the state of the processor and peripherals. These registers have to be identified so that code generators do not attempt to use them as temporary storage.

- **Costs**: Any instruction has a set of costs associated with it, such as the number of cycles it would take to execute the instruction on the hardware, or the number of additional words it may require. The code generator needs to be aware of such costs in order to be able to make trade-off calculations between different implementations of the same piece of code.

- **Cycle-Accurate Simulation**: A form of simulation that assigns execution times (in cycles) to each instruction in the instruction stream, and where these execution times correspond to the exact number of cycles the same instruction stream would take on the real hardware.

- **Cycle Cost**: The number of cycles it would take for a particular instruction to execute on the hardware.

- **DSP**: Digital Signal Processor. A processor specifically designed and optimized to operate on digital data streams. Such processors usually have complex architectures capable of high numerical throughput and rely on hardware acceleration for a lot of functions (such as multiply-accumulate operations, address generation, butterfly operations for FFT etc.). Embedded systems in data-dominated applications (applications where most of the emphasis is placed on data manipulation rather than control flow operations) typically rely on DSPs for their processing power.

- **Data Register**: A register included in the data path for the explicit purpose of performing data manipulation. Such registers do not have side effects when written to, and they return the last value written to them when read from. They act as simple storage elements.

- **Delay Slots**: In pipelined architectures the outcome of a branch instruction may not be available until a few pipeline stages *after* instruction fetch. In this case, instructions following the branch may have already been loaded into the pipeline even though the branch eventually determines control should have been transferred to a different location in the program. Some architectures specify that such instructions will be executed even if control is transferred to a different location in the program. The number of instructions following a branch that will complete execution when control is transferred to the target of the branch, are called delay slots. The code generation should take delay slots into account when emitting branch instructions and either emit the branch instruction correspondingly early in the instruction stream (if possible), or fill them with NOOPs otherwise. Since delay slots complicate code generation and expose implementation details to the instruction set, they are often avoided by flushing or stalling the pipeline, or by the use of branch prediction.

- **Design Criteria**: A set of constraints on the implementation of an architecture such as maximum silicon area, maximum power consumption, minimum performance, etc.

- **Embedded System**: A computer system dedicated to a particular application (such as an engine management computer in a car, or a digital filter in a sound processor). Most computers that are not used for general purpose computation are embedded systems.

- **Exception**: A condition that arises when an operation cannot proceed normally, either because of the current state of the processor or because of the values it received as input. Examples of exceptions are division by zero (which cannot proceed because of the input values), or a privileged mode instruction being issued in normal user mode (which cannot proceed because of the current mode of the processor). Typically exceptions will cause a control flow operation to a fixed address where a small program called an exception handler will try to recover from the failure.

to have binary images where the subsets of bits overlap. This is called binary image overlap and sometimes makes it harder to decode the instruction word.

- **Bitfield Assignment**: The action of setting the appropriate value to the bits in the binary image of an operation.
- **Bitfield Conflict**: In a VLIW instruction it is possible for two operations to attempt to set the same bits in the instruction. This implies that there is a binary image overlap between the two operations. Obviously since the binary image of an operation is the minimum set of bits that must be set to specific values in order to identify an operation, it is not possible for two operations with a binary image overlap to co-exist in the same VLIW instruction. This is called a bitfield conflict. Put simply, it means that two operations in the same instruction are trying to set the same bits to contradictory values and this cannot be allowed.
- **Bitfield**: A contiguous subset of bits in the instruction word.
- **Boolean Clauses**: A set of clauses formulated in Boolean algebra that yield a *true* or *false* answer given a set of *true* or *false* inputs. ISDL uses Boolean clauses as constraints by using operation matches as the inputs.
- **Branch Prediction**: A method of avoiding stalls and delay slots by attempting to predict the outcome of a branch early into the pipeline. The moment a branch instruction is fetched, a piece of logic attempts to guess the outcome of the branch, and hence where the next instruction is going to come from, before it is fetched. Therefore, if the guess is right, no flushing of the pipeline is necessary and no delay slot instructions have to be declared. Pipeline flushes or stalls decrease performance in heavily pipelined machines and delay slots complicate code generation and expose implementation details to the Instruction Set. There are two main methodologies for obtaining the guess: hint-driven in which the branch instruction itself provides the guess and is therefore under the control of the programmer, and automatic in which cached previous outcomes for the particular instruction are used and are therefore probabilistic.
- **CISC**: Complex Instruction Set Computer. Architectures which contain complex and heavily encoded instructions in their instruction sets. Typically they have a variety of complicated addressing modes and instructions that take multiple cycles to complete.
- **Cache Access Pattern**: The sequence of addresses accessed during execution, especially in the context of how this sequence affects any caches present in the system.
- **Code Generator**: A software tool that takes as input a representation of a piece of source code and emits assembly or binary code specific to an architecture to implement that particular piece of source code on that particular architecture. Usually the input is in a compiler intermediate form.
- **Constraint**: A boolean clause with operation matches as inputs, which operate on a current instruction or stream of instructions. If the boolean clause returns a value of *false* then the instruction is in violation of the constraint. This could either mean that the instruction is malformed or improperly scheduled. In either case the hardware cannot guarantee that the instruction stream will execute as expected. If the boolean clause returns a value of *true* then the instruction is not in violation of this particular constraint. It does not imply, however, that it is well-formed and properly scheduled since it might still violate other constraints. Only an instruction that does not violate **any** constraints is well-formed and properly scheduled.
- **Constraints Section**: One of the sections in an ISDL description specifically dedicated to exposing restrictions in the form of operations or instructions, or the scheduling of instructions. These are generally restrictions imposed by the hardware.
- **Control Flow**: Operations that deviate from the sequential fetching of instructions, such as branches, jumps, subroutine calls, returns etc.

A Glossary Of Terms

- **ASIC**: Application Specific Integrated Circuit. An IC custom-designed for one specific application. Usually most custom circuitry in embedded systems is in the form of ASICs.
- **ASIP**: Application Specific Instruction-Set Processor. A processor with an instruction set which is custom-designed for a specific application. Since embedded systems typically have no use for general purpose facilities, they make use of processors with custom instruction sets to reduce the cost of the system and improve performance.
- **Additional Instruction Word**: In some architectures, the instruction word is not long enough to accommodate large constants (such as the destination addresses for jumps and branches) as well as the usual op-codes. In this case, the constant is instead placed in the next instruction word and this word is loaded when needed, but not interpreted as an instruction. This second word is called an additional instruction word.
- **Addressed Storage**: A storage unit that behaves as a group of storage elements, generally of the same width. Individual elements (called locations) may be referred to with an address. Typically, this address is an integer acting as an index into the group of elements.
- **Addressing Modes**: There are various ways for an operation to access its input and output parameters - these are called addressing modes. Most RISC processors provide only a limited number of addressing modes (such as load/store architectures which only offer a register addressing mode). Most DSP and CISC type processors have a wide variety of addressing modes.
- **Ambiguous Instruction Set**: An instruction set with an operation which is multiply-defined and there are at least two of these definitions which will both accept a given instantiation of the operation. For example `Operation 4,6` could correspond to either of the operation definitions below:

```
Non-Terminal ADR:  INT ...  |
                   NAME ... ;
Field Bad:
    operation INT, ADR ....

    operation ADR, ADR ....
```

- **Architecture Exploration**: A methodology for performing architecture design based on iterative improvement. An initial design is created and evaluated for a particular application. Based on measurements made during the evaluation phase, improvements are made to the architecture, either to improve performance or to reduce cost without sacrificing performance. The process is repeated until no further improvements can be made.
- **Behavioral Language**: A machine description language that uses the behavior of operations in the instruction set to describe the hardware. Behavioral languages generally avoid structural information (such as the structure of the pipeline). They are high level languages so they are generally easier for automatic tools to process, and easier for human engineers to work with.
- **Binary Image**: During the assembly phase, each operation in the instruction set will impart a unique combination of values to a subset of the bits of the instruction word. The set of values of the bits and the subset of the bits, together form the binary image of the operation. This subset of bits can be used to uniquely recognize the particular instance of the operation as being in the corresponding VLIW instruction.
- **Binary Image Overlap**: The binary image of an operation consists of a subset of the bits of the instruction word and a set of unique values for these bits. It is possible for two operations

implementation. Future versions of ISDL may contain features to allow the full description of such architectures.

10.2 Co-processor Interfaces

ISDL currently does not support the inclusion of co-processor instructions in the instruction set except as completely specified operation definitions which are indistinguishable from the main processor instructions. However, in a system where ASICs may be involved, it may be beneficial to treat the ASIC portion as a co-processor and not have to fully specify the instructions belonging to this co-processor. This would allow the partitioner to emit assembly instructions to handle the interface to the hardware that was partitioned out, and still allow the assembler to be able to assemble these. At the same time however, it would allow the compiler to ignore these. Future versions of ISDL may support this feature.

10.3 Variable Length Instructions

The current version of ISDL assumes an instruction word of fixed length.[22] While fixed-instruction lengths cover the overwhelming majority of architectures, future versions of ISDL might allow the description of variable-length instructions.

10.4 State-Dependent Constraints

Constraints in the current version of ISDL cannot refer to the current state of the hardware. While this still allows the description of most constraints and a subset of the remaining constraints can be inferred from write hazards in the instruction set, it would be useful for constraints to be able to refer to the current state of the hardware. A number of DSP-style processors contain control information (such as loop counters and termination addresses for zero-overhead loops) embedded in the state. Such processors may require constraints that can refer to these values in order to be able to fully describe all constraints. Future versions of ISDL will allow constraints to refer to the current hardware state.

[22] This is not strictly true since it does allow the description of architectures that require additional words (e.g. to provide long constants that cannot be embedded in the normal instruction fields). Even these, however, can only be an integer number of words. Furthermore, it does not allow for a single instruction to be shorter than usual if some functional units are not being used.

10 Future Work

The current version of ISDL can describe a large number of architecture types. It can also support the generation of the basic set of tools with almost complete functionality. However, it is by no means complete. In particular, a number of features can be added to support an even wider range of architectures, and at the same time provide more functionality in the generated tools. Some or all of these features may be added to future versions of ISDL. This sections describes most of these possible improvements to ISDL.

10.1 Optional Architectural Details

This section is unspecified in the current version of ISDL. However, there are a number of architectural features that might be included in this section in later versions of ISDL:

10.1.1 Exceptions And Interrupts

ISDL currently has no way of describing exceptional conditions in the hardware and how these affect the current state of the machine. This is mainly because the current version of ISDL focuses *only* on actions visible through the Instruction Set and ignores all extraneous events. An artifact of this limitation is the fact that traps (i.e. software generated interrupts) and any operations associated with them are currently supported by ISDL, but hardware interrupts and exceptions are not. This means that the generated simulators will handle traps correctly but will ignore exceptions as if they never happened. Also, the generated simulators have no hooks that hardware-generated interrupts may be tied into. Similarly, the code generators are aware of traps and may generate code using them and/or handling them, but cannot generate code to handle exceptions and interrupts. Such code must be written by hand in assembly, and included in the final stage of compilation.

10.1.2 The Effect Of Caches And Memory Management Hardware

The current version of ISDL only concerns itself with the main processing core. Therefore, the effect of caches and memory management hardware is considered extraneous to the system described by ISDL and cannot be modeled by the generated simulators. Similarly, the code generators cannot perform any kind of memory address pattern optimization since it cannot obtain the required information from the ISDL description. If the interrupt hooks were available (see Section 10.1.1) then the effects of both caches and memory management hardware could be included in the simulators as external modules. However, given that a number of useful optimizations can be performed (especially optimizations on cache access patterns), it might be better to include facilities in ISDL to allow for the description of such hardware features.

10.1.3 Branch Prediction Without Compiler Hints

Branch prediction hardware, unless driven by compiler hints, falls outside the realm of the current version of ISDL.[21] This means that the simulators will no longer yield the same cycle count as the hardware (they can be either optimistic or pessimistic). Furthermore, the opportunity for compiler optimizations for branch prediction hardware is lost. Future versions of ISDL may include features to allow the description of such hardware.

10.1.4 Super-Scalar Architectures

ISDL does not currently provide explicit support for super-scalar architectures. A description can be written for the version of the architecture that has the capability of issuing only one instruction at a time. However, the generated simulator will obviously not be cycle-accurate and the code generator will be incapable of performing optimizations specific to the super-scalar nature of the hardware

[21] Hint driven branch prediction *can* be described in the current version of ISDL since it appears as different instructions in the instruction set with differing timing parameters

Note that both options involve modifying the assembly for the processor. The assemblers generated by ISDL will very often obey a slightly different syntax than the commercial assemblers for the same processor but will always produce the same binary.

One final comment in defining operations: we found it useful to prefix the name of each operation with the name of the field it belongs to avoid name conflicts and reduce name-space clutter.

9.5 Writing The Constraints Section

Most of the issues concerning constraints have already been explained in Section 7.3. Just remember to make all operation matches exact.

The only new thing to mention is the fact that overlap in constraints is neither bad nor useful. In other words, if an instruction violates a constraint, it's wrong. It doesn't matter if it violates one or ten constraints, it is still wrong. Therefore, there's no reason why you should apply any effort to make sure that instructions forbidden by one constraint are not forbidden by any others. Feel free to write constraints that overlap in terms of which instructions get excluded.

Another technique that we found useful was to write matches for the instructions that should be excluded and then reverse them by enclosing them in a negation operation. This way it is much easier to generate exact matches.

Finally, we cannot overemphasize the importance of writing good comments that explain what each constraint does. Because of their form, constraints are very concise representations of the hardware restrictions but they are also almost illegible. You should have a comment accompanying each constraint that explains in plain simple English what restriction the constraint is referring to. If you have the User's Manual for the target architecture, it might be a wise idea to include a reference to the section in the manual where the restriction is explained in detail.

9.6 General Hints And Tips

Finally, here's a set of general comments that might make life easier. We recommend that you write the storage section first. This is almost always the easiest to write and is not often affected by what appears in the other sections. The other sections, however, *are* affected by what appears in the Storage section. We also recommend that you write the Constraints section last. You need to have the definitions for all the operations before you can effectively write the constraints, and the constraints do not affect any of the other sections. We recommend you tackle the Format section next, but only after you have had a good look at all the operations. Keep in mind that changes to the Format section may involve changes in every bitfield assignment written so far, so it would be useful to finalize the Format section before writing any bitfield assignments. Unfortunately, the format of the instruction word really depends on the bitfield assignments so you should be aware of what the assignments look like *before* you start writing the Format section. The Definitions section and Instruction Set section are usually written together, with definitions constantly added as you work on defining the operations.

`= 1`, `Stalls = 2`, `Latency = 1`. Note that the latency of the operation is 1 since the effects of the current operation will be visible to operations in the next instruction.

- A 4-stage pipeline with 2-delay slot instructions. The values for control flow operations are: `Cycle = 1`, `Stalls = 0`, and `Latency = 3`. The third instruction after the current one will be fetched from the target of the branch.

9.4.5 General Comments On Operation Definitions

Finally, here's some general advice on writing operation definitions. First, lets address the problem of ambiguous instruction sets. In Section 9.4.2 we explain what ambiguous instruction sets are. Unfortunately, the instruction sets of some commercial processors are ambiguous. This is because most commercial assemblers are smart enough to perform optimizations that the assemblers generated from ISDL are not smart enough to perform. Let's review the example in Section 9.4.2:

```
Non-Terminal ADR:  INT ...  |
                   NAME ... ;
Field Bad:
    Operation INT, ADR ....

    Operation ADR, ADR ....
```

Like we said earlier, the overlap occurs because of the `INT` token: it is present in both the first definition of the operation, and the second through the non-terminal `ADR`. Why would an instruction set be deliberately ambiguous? The truth is that it is not really ambiguous - the two `INT`s correspond to different types. Consider the case where the first operation definition (which takes an `INT` token directly), accepts only a short integer which can be encoded directly in the instruction word. Similarly, the non-terminal accepts a long integer that needs an additional word to be stored. A commercial assembler will distinguish between the two just by checking the size of the representation needed to encode the integer constant. If it is small enough to be encoded directly in the instruction word, the first form of the instruction is used. If not, the second form is used. In other words, the assembler distinguishes between the two forms of `INT` and considers them as different types.

The problem is that the assemblers automatically generated from ISDL cannot make the same distinction. In ISDL we need to explicitly create a distinction between these two operations. We have two options:

- We can change the operation name for one of the two operation definitions. The instruction set is no longer ambiguous since the two operations now have different names.

- We can add a parameter to one operation, that is just a constant string (or character) denoting which of the two operations is in use. For example:

  ```
  Token short     SHORT {};
  Non-Terminal ADR:  INT ...  |
                     NAME ... ;
  Field Bad:
      Operation short,INT, ADR ....

      Operation ADR, ADR ....
  ```

  Thus, in order to use the first form of the operation you would have to add the string "`short`" as the first parameter. Since the two operations no longer have the same number of parameters, the Instruction Set is no longer ambiguous.

9.4.3 Writing RTL Actions And Side Effects

Since the ISDL version of RTL and its intended purpose is so simple, this should be one of the sections that should not be much trouble to write. There is a few things to keep in mind though. The number one principle is the ever-pervasive *KISS* principle: Keep It Simple Stupid! Avoid excessive complexity - you should not need it. Most operations can only perform very simplistic actions.

One issue is to decide what goes into the action clause and what goes into the side effects clause. The two have almost identical syntaxes and both describe the effects of the operation on the processor state, so where does a particular action go? The first rule of thumb is that if you need access to the special conditionals (see Section 6.1.3) you should put the expression in the side effects clause. Similarly, if you need to inspect the effects of a particular action *after* the action has taken place, the expression should be in the side effects clause. From the remaining expressions, only expressions that describe what the operation was designed for should be included in the actions clause. The rest should be placed in the side-effects clause. For example, an addition operation is likely to have two effects: first it adds two (or more) arguments and stores the result in a destination operand. At the same time it may set the overflow flags and carry flags (and maybe a negative result flag). The operation was included in the instruction set specifically to add arguments (hence the name). The expression describing the addition should be the only one in the action clause. The rest of the effects are just incidental to the addition calculation and should be placed in the side effects clause.

There is one register whose updates need to be handled with care: the Program Counter. The main reason for this is that the code-generator needs to be able to recognize some updates to the PC as operations that are useful for flow control. Also note that the PC is auto-incremented as a side-effect to each *instruction*.

9.4.4 Writing Up Cost And Timing Expressions

There is one technique that makes cost expressions easier to write: pick one of the fields as the primary field and make the cost of this field be the cost of the complete instruction. Then add to this additional costs (if any) from the other fields. To calculate the cost for each field, each operation should have a constant cost (reflecting the actual cost of the operation), and add to it further costs from the non-terminals in the operation (since various options in the non-terminals may increase costs). This is hard to explain without the use of an example so we recommend you look carefully at how the cost calculations are performed in the Motorola 56000 description in Appendix C.

The only exception to the above rules is the `Stall` cost which should be constant for all operations. ISDL does not restrict this cost to be constant, but it would take a very weird architecture to require a non-constant `Stall` cost.

As for the timing parameters, these are usually constant and only depend on the operation at hand, so it should be relatively straightforward to come up with the right expressions for them.

In order to clarify how these parameters interact, and how they can be used to describe pipelines in the architecture, consider the following example architectures:

- A 4-stage pipeline with no bypass logic and no stalls (completely unprotected pipeline). The values are: `Cycle = 1`, `Stalls = 0`, and `Latency = 4`. This means that the architecture can issue one instruction per clock cycle, it will never stall, and the effects of any operations will be visible to the fourth instruction after the operation is fetched.

- A 4-stage pipeline with full bypass logic and no stalls (completely protected pipeline). The values are: `Cycle = 1`, `Stalls = 0`, and `Latency = 1`. This means that the architecture can issue one instruction per clock cycle, it will never stall, and the effects of any operations will be visible to the instruction immediately following the operation (because of the bypass logic).

- A 4-stage pipeline with full bypass logic except on loads where two stall cycles are necessary (completely protected pipeline). The values for all operations other than the load operation are: `Cycle = 1`, `Stalls = 0`, and `Latency = 1`. The values for the load operation are: `Cycle`

operations, one from each field. It is therefore a good idea to define one field for each functional unit in the architecture. Generally this will be enough. However, in cases where one functional unit generates a result which is used by another functional unit in the same clock cycle, it is extremely hard to create an RTL action clause which properly describes the required effect. Consider, for example, the case of an address generator whose output is fed straight into the address port of a data memory. Now consider the case where the address generator is controlled by one field and the memory operation by another. There's no way to write the RTL for this without using temporary value holders and ISDL does not allow the use of value holders. There is an even bigger problem with splitting the RTL action over two or more operations: code-generators have a hard time dealing with RTL actions that are split in this manner.[20]

The first way to circumvent the problem is to absorb the operation into the field of one of the two functional units. For example, you could make the load and store operations be part of the field that describes that address generator functions. Or alternatively we could make them part of field that describes memory operations. In either case, since we are using both functional units in a single operation we should make sure that we include the appropriate constraints. For example, if we made the operation part of the field controlling the address generator functions, we should put a constraint in place preventing the use of any other memory operations when the particular operation is in effect.

There is another way to circumvent the problem. We can define a whole new field. Conceptually, this is cleaner since it groups together only operations that are similar in nature and form. Just like in the previous case, we have to put constraints in place that will prevent the use of the address generator and the memory for other purposes while this operation is in effect. The only disadvantage to this approach is that it makes the assembly instructions longer. It should therefore only be used if there are a number of operations that can be included into it and those are used more than sparingly.

In general, you should probably define one field for each functional unit, and maybe more for special circumstances.

9.4.2 Multiple Operation Definitions

One feature that was not mentioned so far is the provision for an operation to have multiple forms. In other words, it is possible for multiple operation definitions to share the same operation name but have different numbers and/or *types* of arguments. Providing multiple definitions for the same operation is a way of avoiding non-terminals that contain very complex RTL action and side effect clauses and avoiding the inclusion of bitfield assignments in non-terminals. The only problem with multiple operation definitions is that the designer has to make sure that multiple operation definitions are truly different in terms of the type and/or number of parameters. Consider the following case:

```
Non-Terminal ADR:  INT ...  |
                   NAME ... ;
Field Bad:
    Operation INT, ADR ....

    Operation ADR, ADR ....
```

The above is not a valid example of multiple operation definitions since the instance "`Operation 4,6`" could belong to either definition. In this case we say that the instruction set is ambiguous. Section 9.4.5 gives a solution to the above problem; for the moment we shall emphasize that you should not write up ambiguous instruction sets in ISDL. You can ensure that this is the case by making sure that any multiple operation definitions have a different number of parameters or have parameters of different types.

[20] In fact, this is the reason why ISDL does not support the use of temporary value holders: to force designers to avoid splitting RTL actions.

("+" and "*" respectively) should be avoided if possible since they are harder for the disassembler to decipher than the bit manipulation operators. In other words, you should use "`(x << 1)`" instead of "`(x * 2)`". Also avoid using values of already assigned bitfields in calculating the values of a new bitfield unless absolutely necessary. In other words avoid expressions of the form "`x = x + 5`" if at all possible. The best form for bitfield assignment expressions does not contain any arithmetic operators and does not have bitfield names on the right hand side.

9.2.5 Non-Terminal RTL Actions And Side Effects

Most of the time, it is relatively straightforward to decide what should be in the action and side effects clauses. Most non-terminals will be used to describe addressing modes anyway, in which case the RTL expression for both the action and the side effects clauses should be the storage reference that describes what storage is referred to by the given addressing mode option. Sometimes things will be a little more complex, and a full assignment statement may be required in the non-terminal definition. These should be avoided if possible. Defer assignments to the operation definitions to allow maximum reuse of the non-terminals. Conditional RTL statements should *always* be avoided in non-terminal definitions - if you have to include conditional RTL statements in a non-terminal definition you are probably trying to do too much with a single non-terminal. You should probably consider multiple operation definitions instead.

9.2.6 Non-Terminal Costs And Timing

Costs and timing expressions should also be relatively straightforward to write for non-terminals. Most of the time the values will be zero. In some occasions (especially in the case of non-terminals representing addressing modes) there may be values other than zeros. You should avoid using field names in the cost and timing expressions of non-terminals to maximize reuse.

9.3 Writing The Storage Section

The storage section is usually relatively simple. There's simply very little freedom as to how the storage resources are described. The only issue arises with the use of aliases.[19] The rule of thumb is that aliasing should only be used in two cases:

- In the case where storage from different units (for example different memories or different register files) must be combined into one unit (or respectively split one unit into a number of different units).
- In the case where an addressed storage unit has two or more forms (for example, a 3-register register file used as a single long register in special occasions). Note that this is really a degenerate example of the first case but since it is much more common it deserves special mention.

9.4 Describing The Instruction Set

The Instruction Set section is the other part of ISDL description with many degrees of freedom. There are choices to make as to the subdivision in fields, how many forms of each operation should be defined, as well as multiple issues concerning each part of an operation definition. These are examined in the following sections.

9.4.1 Dividing The Instruction Set Into Fields

This is one of the key decisions that need to be made, but unfortunately it is not as straightforward as it first seems. Recall the definition of a field: it is a set of instructions that are mutually exclusive (i.e. they can never appear together in a VLIW instruction). A VLIW word is made of a set of

[19] These represent the only degree of freedom in the Storage Section.

- The name of *any* operation defined in the instruction set
- Labels for addresses in the instruction stream

Typically, tokens are used to define register names and constant character strings. In the case of names of registers that belong to a register file, it is wise to create a single token that covers all the registers in that register file by using a common string prefix to identify the register file and a number suffix to identify the register in the register file. Another example taken from the sample SPAM VLIW description:

```
Token   "AG1.R"[0..7]     AG1_R   { [0..7]; };
Token   "AG2.R"[0..7]     AG2_R   { [0..7]; };
```

In this case the return value identifies the number of the register in the register file.

There is only one trick to pass on when it comes to defining tokens: how the return values are specified. If a token is grouping together multiple entities (as in the above example), then it is likely that the binary image of the operation that uses it will contain some form of encoding of the identifiers of the entities (in this case the register index). In such a case, it is wise to arrange for the return value to be as close as possible to this encoding, thus simplifying the bitfield assignment definition of the operation.

9.2.3 Creating The Non-terminal Definitions

Non-Terminals are the main source of abstraction in ISDL. Most of the degrees of freedom in ISDL come from non-terminal definitions.[18]

This of course means that non-terminals are the most important parts of an ISDL description when it comes to readability and conciseness. The rule-of-thumb of following the Programmer's Manual applies just as strongly to non-terminals as to everything else. Usually, the Programmer's Manual will already contain the best abstractions described as tables. For example, the Motorola 56000 Programmer's Manual already contains most of the non-terminals defined in the 56000 description in a set of tables. You may have to define some more to cover special cases but in general you should be able to follow the manual closely.

There are a number of things to pay attention to when you are defining non-terminals. The first is when to define a non-terminal. The alternative is to define multiple instances of the same operation (one for each option of the non-terminal). These take the form of multiple operation definitions where the operation name is the same. Given the overhead in defining an operation, if in doubt you should probably define a non-terminal instead. *If in doubt, make it a non-terminal.* The only problem with defining too many non-terminals is name space cluttering. If you give your non-terminals meaningful names this should not be a problem.

9.2.4 Non-Terminal Return Values And Bitfield Assignments

Another thing to keep in mind is the tradeoff between using return values instead of bitfield assignments in the return value clause of a non-terminal definition. The tradeoff is really between re-usability of code and levels of indirection in performing the bitfield assignments. Our recommendation is to avoid bitfield assignments in non-terminal definitions unless absolutely necessary. If in doubt, use a return value encoded in the right way. The return value approach allows non-terminal definitions to be used in many more operation definitions (and other non-terminal definitions) by deferring the bitfield assignment to the actual operation definitions. Given that the purpose of ISDL is to support architecture exploration, re-use of ISDL components is vital in minimizing the effort of making small architectural modifications and evaluating them.

We should also mention that the style used in writing bitfield assignments and return values seriously affects the performance of the disassembler. The addition and multiplication operators

[18] The other main source of abstraction is the operation definitions themselves. It is no accident that operation definitions and non-terminal definitions are so similar.

their own. You will have to perform bit manipulation operations to set these anyway. The trick is to avoid spilling these bit-manipulations into subfields that do not need them simply because you included too many bits in the subfield. If in doubt, it is better to split a word or field into too many subfields than into an insufficient number; if nothing else you can always write up a split function for the cases when the bits should actually be grouped together.

In terms of correctness requirements, there is only two things to remember: *all* bits in the instruction word must be part of one and only one subfield, and the subfields should appear in the order in which their constituent bits appear in the instruction word.

9.2 Writing Up Global Definitions

The definitions section is where most of the abstraction in ISDL takes place, so it is the one that allows you the most degrees of freedom. Fortunately there's not many things to watch out for in writing up these definitions and almost any mistake in form can be corrected by writing even more definitions. There is, however, a big difference in readability between descriptions that have a lot of almost similar definitions in them, and the minimalist ones that have only the amount of abstraction that is necessary.

Since the three types of definitions (tokens, non-terminals, and split-functions) are almost completely independent we shall describe them individually.

9.2.1 Creating Split Functions

In essence, split functions reverse the divisions that the format section imposes on the instruction word. Often there will be an operation that wishes to include a large binary constant into the instruction word. This is usually a constant that was provided as one of the parameters. Typically the operation will take over a large number of subfields (usually contiguous) and use them as a long binary string to hold the constant. Without split functions this would complicate the bitfield assignment definition for the operation unnecessarily. The bitfield assignment statement would effectively break up the constant into the subparts that correspond to each of the subfields, then shift each of these subparts to align them, and then assign them to the right subfields. Instead you can perform exactly the opposite procedure. You can use a set of subfields (not necessarily contiguous) to form a longer bitfield with width equal to the collective width of all the subfields. This new bitfield can be given a name. Then you can set this new bitfield the same way you would set any other subfield, so in the case of a large constant you can just assign the constant to the subfield using the appropriate function.

In any case where you find yourself trying to split a constant into two or more subfields you should consider the use of a split function. There is, however, one condition that must be satisfied first: When you use split functions, you set every single bit of every single subfield in the split function definition. If your intention was not to set all bits of a certain subfield then you should either avoid the use of a split function or split the given subfield into two or more subfields and only include the ones you want in the split function. Therefore, it may be better to forgo the use of a split function in some cases, than to subdivide a subfield into too many parts. If in doubt though, you should probably use the split function, since even when you divide a bitfield into too many subfields you can still create yet more split functions to join these parts back up. This is an example where more definitions can correct for earlier mistakes.

9.2.2 Coming Up With The Token Definitions

The token definitions are probably the easiest statements to create in ISDL. Their existence is determined almost exclusively by the assembly syntax of the instruction set and leaves very little room for interpretation. The rule for writing token definitions is as follows: if it is part of the assembly for any operation and it is not one of the pre-defined types, it should be defined as a token. The pre-defined types are:

- Numerical constants (in decimal, hexadecimal or floating point notation)

9 Hints And Tips On Writing Good ISDL Descriptions

While ISDL does not contain an excessive set of features, it does provide some powerful abstraction mechanisms. This means that there are multiple ways to describe any single architecture and sometimes the best one is not necessarily the most obvious. This section is intended to help human designers develop the art of writing small, concise and easy to understand descriptions of architectures. We provide a set of hints and tips for each of the sections of an ISDL description and then a set of general suggestions that might apply to all sections.

Before we start though, there is one general hint that you should always keep in mind: If you are describing a commercial architecture for which you have a programmer's manual then *use it*. Typically, you will find that the people that wrote the manual spent a lot of time and effort trying to express things in as clear a manner as possible and make all the appropriate abstractions. Usually, you will not need to do much more than just follow the structure of the manual. Don't forget that ISDL is designed to be as similar to the programmer's manual as possible.

9.1 Writing The Format Section

The main degree of freedom in writing a description of the instruction word is the way it is divided into subfields (the division into fields is mainly to assist in naming). Coming up with the right division can make the task of writing bitfield assignments a lot easier. If you have a programmer's manual for the architecture you are describing this will usually contain one or more subdivisions. These are usually excellent starting points. If not, then you will have to come up with your own.

In a VLIW architecture, each unit will generally have a set of bits that provide the op-codes and parameters for the unit. There might be some overlap with other units on rare occasions but there should be a very obvious division of the instruction word into different parts that correspond to the various units. If you find such a division then make these the fields and name them after the unit they control. Look at the example below (taken from the sample SPAM VLIW):

```
Control = OP[4], RI[6];
AG2     = OP[4], RW[3], RA[3], RB[3], RMEM[3];
AG1     = OP[4], RW[3], RA[3], RB[3], RMEM[3];
ALU     = OP[4], RW[5], RA[5], RB[5], RMEM[5];
MAC     = OP[3], RW[4], RA[4], RB[4], RMEM[4];
DB      = SRC[4], SINK[4];
ALUOP   = OP[2];
MACOP   = OP[2];
DM1A    = OP[1];
DM2A    = OP[1];
```

As you can see, there is generally a clear division of bits between the functional units. Occasionally, the Control field will borrow the AG2 field bits for constants, and the DB field will borrow both the AG1 and the AG2 fields, but other than that, the division stands for almost all operations. Also note that the fields were named after the units they control.

Once you have determined the division into fields then you are left with the task of subdividing them into subfields. The method is almost the same. If you closely inspect all the operations that use a given field (and typically all such operations will belong to a single functional unit), you will notice that there is yet a finer division into subfields according to what the bits in each subfield are used for. For example, each functional unit will need an op-code so in most cases each field will have a certain number of its bits used as an op-code. You may split this subset into a subfield of its own. A similar observation can be made for the identifiers of the various parameters of an operation. Addressing modes usually will also use the same set of bits each time they appear. These are good candidates for subfields. Some fields may only have one subfield. Again, subfields should be named with names that are mnemonics for their function.

On occasion, some sets of bits may be used by so many different and unrelated operations that there is no clear-cut place for them in any of the fields. These are best split out into a subfield of

8 Optional Architectural Information

The current version of ISDL does NOT predefine any Optional Architectural Information. Various tools that use ISDL should not define their own since the format for such information may change at a later time.

For a discussion of a number of features that might appear in this section in later versions of ISDL see Section 10.1.

The expression inside the parentheses will return true in the case of operations like "`DB_Move AG1.R2, AG1.R7`" where both parameters contain a dot somewhere and the portion before the dot is the same in both parameters. An examination of the syntax of the "`DB_Move`" operation will show that this is equivalent to saying that the "`SRC`" and "`DEST`" parameters refer to the same register file. The "`~`" makes the constraint false if the expression is true meaning that any instruction that contains such a malformed version of the "`DB_Move`" operation would be invalid.

ISDL also allows constraints to express conflicts between operations occurring in different instructions. In the Motorola 56000, for example, an instruction containing a "`Main_DO`" operation cannot immediately follow an instruction containing a "`Main_REP`" operation. The following constraint expresses that restriction.

```
~((Main_REP *) & [1](Main_DO *))
```

of the operations. These expressions must be *exact*, i.e. they must include all forms of the operation that are relevant but not include any that are not. It may be necessary to use the "|" and "&" operators to ensure that operation matches are exact. Suppose, for example, that the following operations are available:

```
MUL RA, RB
MUL RA, RB, RC
Move X,Y
```

Additionally, the hardware imposes the restriction that a two-address "MUL" (i.e. the first instruction in the above example) cannot be used in the same instruction as a "Move" operation. First, operation matches for the two operations must be constructed. The match for the "Move" operation is trivial - one can simply use "Move *". The match for the "MUL" instruction is not as simple. "MUL *" is incorrect as it would match both forms of the "MUL" instruction. "MUL *,*" would also match both forms of the "MUL" instruction.[16] We somehow need to describe an expression that excludes the second form of the "MUL" instruction. This can be achieved by explicitly removing that particular match as follows:

```
(MUL *) & (~(MUL *,*,*))
```

In this case , the first match will match both forms of the "MUL" instruction and the second match will match only the second form of the "MUL" instruction. Since the first match must be true and the second match must be false (because of the "~") for the overall expression to be true, the complete expression will be true if and only if we encounter the first form of the "MUL" instruction.

Once expressions that match each of the operations have been created, the next step is to combine them in a form such that the whole expression is false if both operations appear, and true otherwise. This is equivalent to saying that the resulting expression should be false if both component expressions are true, and true otherwise. Hence, in the above example the resulting constraint would be:

```
~((MUL *) & (~(MUL *,*,*)) & (Move *))
```

Note that sometimes it is possible to create exact expressions without using anything other than the wild-card operators. If, for example, the following forms of the "MUL" instruction are available:

```
MUL R[0-7], R[0-7]
MUL M[0-7], R[0-7]
MUL M[0-7], M[0-7]
```

and it is necessary to match only the first form, one possible expression would be:

```
(MUL R*,R*)
```

Some types of constraints may require that two parts of an instruction are (or are *not*) the same. Consider the following example from the SPAM sample VLIW:

```
DB_move SRC, SINK
```

where "SRC" and "SINK" can be any of a number of possible options (for example, "AG1.R[0-7]", "AG2.R[0-7]", "ALU.R[0-31]", and "MAC.R[0-15]"[17]). There is only one port connecting each of the units to the data bus, so the hardware cannot move from one register in a register file to another register in the same register file. Consider the following proposed constraint:

```
~(DB_Move @[1].*,@[1].*)
```

[16] This is because the second "*" would consume as many characters as it can so it could consume either one or two parameters.

[17] These correspond to the to register files for the two address generators, the ALU unit, and the MAC unit respectively.

not one of the form "`Jump-2`". In the case of regular expressions that can match a variable number of characters, there is also the issue of where one expression is considered to end and where the next one begins. These expressions are the "`*`", "`+`" and the variable matches (not the variable references). We use the same rule that regular expressions use: we use the longest possible match for the wild-card expression that can still make the next expression match (assuming there is a way to make both of them match). Consider the following example: the expression is "`*[abc]`" and the operation string is "`Jumpacc`". In this case the two constituent expressions are "`*`" and "`[abc]`". The "`*`" expression will "consume" or match the first 6 characters of the operation string (i.e. "`Jumpac`") while the second expression will match the final "`c`". If, however, the expression was "`*[ab]`", then the first expression would only match "`Jump`" and the second expression would match the "`a`".

- The unary operator "`~`" followed by a boolean expression enclosed in parentheses "`()`". This expression is true if the input expression is false, and false otherwise.

- An input boolean expression followed by the binary operator "`|`" followed by a second input boolean expression. The whole set is enclosed in parentheses "`()`". This expression is true if either of the input boolean expressions is true, and false otherwise.

- An input boolean expression followed by the binary operator "`&`" followed by a second input boolean expression. The whole set is enclosed in parentheses "`()`". This expression is true if both the input boolean expressions are true, and false otherwise.

- An integer n enclosed in "`[ ]`" and followed by an expression enclosed in parentheses "`()`". This expression is true if the input expression is true for the nth instruction after the current one. For example, the expression:

  ```
  (DB_move *,*) & [1](DB_move *,*)
  ```

  would be true if the current instruction contains a "`DB_move`" operation and the following instruction contains a "`DB_move`" operation as well. We call such expressions *time-shifted expressions* and the constraints containing them *time-shifted constraints*.

7.3 Writing Constraints

Before the details of how to write constraints for specific architectures are explained, an understanding of how constraints are used would be helpful. Conceptually, constraints are applied to an instruction stream one instruction at a time. Thus, there is always a pointer which points to the current instruction in the stream. Each constraint is matched against the instruction at the pointer (unless it is part of a time-shifted expression in which case it is matched against the instruction at the appropriate offset from the instruction pointer. If the constraint returns true then the instruction does not violate this particular constraint.[15] If the constraint returns false, then the instruction (or an operation within it) is considered to be invalid. Further investigation may reveal why and what can be done to rectify the situation. Note that constraints that do not involve time-shifted expressions are only matched against the instruction at the pointer so only involve one instruction.

The first step in writing up the constraints section is to identify all hardware restrictions in the architecture. Typically these will either be listed in a section of their own in the Programmer's Manual, or be listed along with the relevant operation description in the Instruction Set portion of the Programmer's Manual. Either way, they should be easy to obtain from the documentation if a commercial architecture is being described. If the architecture is custom, then the architect or the design engineers should be able to compile a list of all relevant restrictions.

Most of these restrictions will be of the form that one operation cannot be used in the same instruction as another operation. In this case, the first step is to make expressions that match each

[15] This does not mean that the instruction is valid since it may still violate some other constraints.

```
AG2.RMEM = CONST ;
AG2.RW = AG2_RW ; }...
```

In the above example, `Control_br` uses the split function for `Split.OFFSET` which sets all subfields of the `AG2` format field. At the same time, the `AG2_addbc` operation also attempts to set these subfields. Only one of them can possibly succeed, so an instruction combining these two operations is illegal.

3. **Lexical Conflicts**: These are not true conflicts in terms of resources - they just express consistency constraints for the assembly of the processor. For example, a close look at Figure 4 will reveal that it is possible to write the value of a register from the `ALU` register file to the memory and to the bus at the same time. There is no hardware conflict that would prevent this. However, the register which would be accessed corresponds to the subfield `ALU.RMEM` and both the memory save operation and the data bus move operation would attempt to set this bitfield. This is possible as long as the same value is written into the bitfield - but this must be enforced by a constraint.

7.2 Constraint Syntax

The constraints section consists of a series of constraints, each one of which is a boolean expression. A boolean expression may consist of the following options:

- An operation match enclosed in parentheses: This consists of a string with special purpose wild-cards embedded in it. The value of this is expression is true if the string (with wild-cards included) matches the textual representation (i.e. the assembly for) an operation in the instruction under inspection. Wild-cards work in the same way as regular expressions except for the fact that the symbols recognized are more similar to the UNIX type shell wild-cards than those of regular expressions.

  An operation match (regular expression) can be one of the following:

  - A constant string: This is a sequence of ASCII characters that forms a string. If special characters need to be included in a string they must be preceded with a "\". Strings match if any part of an operation string is identical to the string.
  - "`?`": This is a special character that matches a single character in the operation string.
  - "`+`": This is a special character that matches one or more characters in the operation string.
  - "`*`": This is a special character that matches zero or more characters in the operation string.
  - A range operator: This consists of a set of characters enclosed in "`[ ]`" and matches any single one of those characters. Ranges such as `[abcdefghijklmn]` may be abbreviated to `[a-n]`. If the first character in the range is "`^`" then it is not considered part of the range and the range matches any single character **not** in the range.
  - A variable match of the form `@[<int>]` where *int* is an integer from 0 to 9. The first occurrence of a reference to variable match x (i.e. the variable `@[x]`) will match zero or more characters from the operation string and store them in a variable. Any subsequent references to this variable in the context of the same constraint will only match the sequence of characters stored in the variable.
  - An expression followed by another expression: This will match if the first expression matches part of the operation string and the second expression matches part of the operation string *starting from where the first expression left off*. For example, the expression "`[a-z][0-9]`" will match any operations that have a lower-case letter followed by a digit anywhere in the string. Therefore it would match an operation of the form "`Jump2`" but

7 Constraints

As was explained earlier, ISDL assumes that a complete VLIW instruction will be formed by choosing one operation from each field, and concatenating all operations in the order the corresponding fields are declared. In other words, instructions are formed by combining operations. However, not all such combinations are guaranteed to be valid. Furthermore, an operation itself may contain a combination of parameters and not all combinations allowable are guaranteed to make sense to the hardware. Consider the example of the data bus move operation in the SPAM sample VLIW:

```
DB_move    SRC,SINK   { } { SINK <- SRC ; }
```

This instruction will move data between various storage units using the data bus. Note, however, that each unit is connected to the data bus through a single port (see Figure 4) so each unit can be either a source, or a sink, but not both at the same time. The way the operation is defined though (through the use of the `SRC` and `SINK` non-terminals), it is possible to create instructions of the type "`DB_move AG1.R2, AG1.R6`" which make the Address Generator 1 register file the source **and** destination of the data bus move. This is in illegal operation. Also note that other operations such as `DM1_bus_save...` also use the data bus and therefore can never be combined with the `DB_move` operation; that would result in an illegal instruction.

Therefore, it is possible for the operation definitions to define operations and instructions which are illegal and which cannot be executed by the hardware. Such combinations must be identified so that the tools will avoid using them. In particular, the code-generators must avoid generating illegal operations or instructions.

In order to make it easy to identify such invalid operations and instructions, ISDL supports the definition of constraints. Conceptually, these are nothing more than boolean clauses applied to instructions or groups of instructions. If even a single constraint is violated, then some operation or instruction is illegal. Knowledge of the constraint that was violated allows one to determine which operation or instruction was illegal. The offending operation or instruction can then be changed accordingly so that the code is now correct. The way constraints are used assumes that there is a current instruction under scrutiny and each constraint is "matched" to it. If the constraint returns false, the instruction is illegal (or an operation within it is). Note also that some constraints may involve instructions other than the current instruction as well.

7.1 Types Of Conflicts

There are three different types of conflicts that can result in illegal operations or instructions:

1. **Resource Conflicts**: In this case, two conflicting operations (or even the same operation at times), attempt to make simultaneous use of a single resource. For example, in the illegal operation "`DB_move AG1.R2, AG1.R6`", the conflicting resource is the data bus port on the Address Generator 1 register file. In the case of an instruction using both a `DB_move` operation and a `DM1_bus_save` operation, the conflicting resource is the bus itself. Most conflicts are of this type and most constraints result from conflicts of this type.

2. **Bitfield Conflicts**: In this case, two conflicting operations attempt to set the same subfield in the binary format of the word. Obviously this is not possible so any attempt to do it is illegal. Consider the following example (taken from the SPAM sample VLIW description):

```
Split.OFFSET    AG2.OP+AG2.RW+AG2.RA+AG2.RB+AG2.RMEM;
...
Control_br OFFSET   { Control.OP = 0x9 ;
                      Split.OFFSET = OFFSET ; }...
...
Field AG2:
AG2_addbc   AG2_RB,CONST,AG2_RW { AG2.OP = 0x0 ;
                                  AG2.RB = AG2_RB ;
```

have this form. Since costs are defined on a per-operation basis, it is possible to have operations in field `x` which include the cost of field `y`, while at the same time having operations in field `y` which include the cost of field `x`. Definitions such as these (which contain cycles in the fields they refer to) are not always well-defined and are not allowed in ISDL. While ISDL does not have any constructs which prevent someone from writing specifications containing cycles in their costs, the tools that process ISDL descriptions are not guaranteed to attempt to make sense of such descriptions. Tools and engineers that produce ISDL descriptions should themselves make sure that they do not produce cost descriptions that contain cycles.

Also note that since timing definitions use the same construct, they are subject to exactly the same comments.

Finally, note that even though ISDL does not restrict any cost to be a constant, the `Stall` cost of all operations will probably be constant.

then the costs will be `Cycle = 4` and `Size = 2`. Note that while this makes it possible to express in an unambiguous way the costs of the whole instruction rather than just operations, proper use of this facility requires caution (see Section 6.2).

In order to arrive at the cost of the final instruction, ISDL simply takes the maximum of the costs of the constituent operations.

6.1.7 Timing

Instruction timing is described using exactly the same expressions as costs, except different names are used for the variables being defined. ISDL requires two predefined timing variables: `Latency` and `Usage`. `Latency` declares how many instructions from the current instruction the results of the current operation will be available. For example, a latency of 1 means that the result of the current operation will be available for use by any operations in the next instruction. `Usage` declares how many instructions after the current instruction a certain "functional unit" (i.e. field in the instruction) becomes available again. For example, a usage of 1 implies that the next instruction can contain operations from the same field as the current operation. A usage of 2 means that the "functional unit" may not be used in the next instruction and therefore a `NOP` should be selected from the corresponding field in the next instruction. If any other operation is selected instead, the hardware will stall the next instruction until the functional unit becomes available once again.

Note that each operation in ISDL has its own timing parameters and there is no unique set of timing parameters for the complete instruction. Therefore, ISDL does not use the maximum of the timing parameters of each operation as the timing parameter of the instruction as a whole.

6.2 Other Operation Definition Issues

Section 6.1 gave a comprehensive description of how operations may be defined. This section examines some related issues.

6.2.1 The Difference Between Actions And Side Effects

It was mentioned earlier that the syntax used to describe the actions and side effects of an operation is almost identical. It was also mentioned that both are used to describe the changes of state that occur when an operation is executed and therefore are very similar in a lot of respects. There is, however, one very important conceptual difference. Actions can be considered to be the desired effects of each operation. These are the effects on the state that the operation is used for, and these effects are the reason why it was included in the instruction set. The side effects are just other actions that handle the book-keeping tasks of the operation. They are not the reason for including the operation in the instruction set - they are simply there to clean up any loose ends left over from the execution of the operation. Consider for example an arithmetic add operation. This operation changes two pieces of state. First it adds two input parameters and saves the resulting value in some storage resource. Then, it checks to see if an overflow resulted and if so sets a special flag. The first is the action of the operation - the second is merely a side effect. In terms of the syntax and RTL structures they use though, the two statements are indistinguishable. This conceptual separation is more than just a convention to make descriptions easier to read. Some tools use this distinction to perform their tasks. For example, the code-generator will usually only use the RTL action portion of an operation definition when it is trying to determine what an operation does.

6.2.2 Issues Related To Cost And Timing Descriptions

In Section 6.1.6 it was shown how we can transfer the costs of an operation from one field to another. This makes it possible to define a master field which sums up the costs of all other fields to derive the cost of the complete instruction. Usually there is a dominant field in the instruction set that can be used for this purpose; if one cannot be found, any field can be arbitrarily used as the master field. Unfortunately, this also implies that caution must be exercised when defining cost expressions that

- An input expression followed by the binary operator "<", followed by a second input expression. The value of such an expression is 1 if the value of the first expression is less than the value of the second expression and zero otherwise.
- An input expression followed by the binary operator ">", followed by a second input expression. The value of such an expression is 1 if the value of the first expression is greater than the value of the second expression and zero otherwise.
- An input expression followed by the binary operator "<=", followed by a second input expression. The value of such an expression is 1 if the value of the first expression is less than or equal to the value of the second expression and zero otherwise.
- An input expression followed by the binary operator ">=", followed by a second input expression. The value of such an expression is 1 if the value of the first expression is greater than or equal to the value of the second expression and zero otherwise.
- `max(x,y)` where `x` and `y` are expressions. The value of this expression is the largest of the values of `x` and `y`.
- `min(x,y)` where `x` and `y` are expressions. The value of this expression is the smallest of the values of `x` and `y`.
- A storage reference. This must be a single register type storage reference and its value is the two's complement integer represented by the contents of the storage.
- The name of a non-terminal. The value of such an expression is the same as the value assigned to the corresponding cost in the non-terminal definition (see Section 4.2.1).
- An input expression enclosed in parentheses, "`()`". The value of such an expression is the same as the value of the input expression. Parentheses are used only to denote precedence.
- The name of a field: Each instruction in ISDL is considered to consist of a number of operations; one chosen from each field. The cost value of interest is not the cost of individual operations (which has no real meaning in terms of what the hardware will do), but rather the cost of the whole instruction. This, of course, may depend on which operations were selected from each field. The value of a cost expression consisting of the name of a field is equal to the cost of the particular operation that was selected from that field. Consider the following example (taken from the Motorola 56000 description):

```
Field Main:
    Main_ABS ACC    ... { Cycle = 2 + DBM; Size = 1 + DBM; } ...
...
Field DBM:
    DBM_Move SIMMD, LAGREGS ... { Cycle = 0; Size = 0; } ...
...
    DBM_Move LIMMD, LAGREGS ... { Cycle = 2; Size = 1; } ...
```

Note that both cost definitions in the `Main_ABS` command depend on the value of the corresponding costs defined in the `DBM` field. These in turn depend on which of the `DBM` operations is selected and therefore the final form of the instruction. If the instruction is of the form

```
Main_ABS ...; DBM_Move SIMMD,...;
```

then the costs will be `Cycle = 2` and `Size = 1`. If on the other hand the instruction is of the form

```
Main_ABS ...; DBM_Move LIMMD,...;
```

The above operation performs a parallel move. If limiting occurred during the move, the `L_F` flag has to be set. The side effects clause in the above operation definition checks the appropriate special conditional and sets the flag if necessary.

Note that there is one special side effect that is implied by all ISDL operations: the increment of the Program Counter. This happens automatically with a latency of 1 irrespective of what is specified in the timing parameters for the operation. However, if the Program Counter is modified inside the RTL action or side effects clause, then the effect of these clauses overrides the auto-increment operation.

6.1.6 Costs

ISDL can define multiple costs for each operation. There are three costs that are predefined by ISDL and are present in any description: `Cycle`, `Stall`, and `Size`. Additional costs may be included in a description for some tools. For more details on which costs are accepted by a given ISDL-based tool consult the tool documentation. Of the predefined costs, `Cycle` attempts to express the cost in runtime of a particular operation. It denotes the number of cycles it would take for the operation to execute on the hardware. The `Stall` cost, describes how many additional cycles will be needed because of stalls inserted to handle data dependencies. For example, a `load` operation might normally require only one cycle to complete, but in the special case where the next instruction attempts to use the value just loaded, it might stall the pipeline for an additional 2 cycles.[14] The cost `Size` expresses the cost of the operation in terms of code-size. It gives the total number of words required for the operation. Each non-terminal and each operation definition have a cost definition associated with them for each type of cost expected by the tools. If there is no definition provided for a certain cost, the value is assumed by default to be 0. The definitions are enclosed in "{}". These definitions are assignments of the form:

```
<cost> = <expr>;
```

where *cost* is the name of the cost being defined (e.g. `Cycle`) and *expr* is an expression. An expression may be one of the following options:

- A constant integer.
- The unary operator "`-`" followed by an input expression. The value of such an expression is the same as the negative of the value of the input expression.
- An input expression followed by the binary operator "`+`", followed by a second input expression. The value of such an expression is the sum of the values of the input expressions.
- An input expression followed by the binary operator "`-`", followed by a second input expression. The value of such an expression is the same as the value of the first input expression minus the value of the second input expression.
- An input expression followed by the binary operator "`*`", followed by a second input expression. The value of such an expression is the multiple of the values of the input expressions.
- An input expression followed by the binary operator "`/`", followed by a second input expression. The value of such an expression is the same as the value of the first input expression divided by the value of the second input expression. The second expression is not allowed to be 0.
- An input expression followed by the binary operator "`%`", followed by a second input expression. The value of such an expression is the same as the value of the first input expression modulo the value of the second input expression. The second expression is not allowed to be 0.
- An input expression followed by the binary operator "`==`", followed by a second input expression. The value of such an expression is 1 if the values of the first and second expression are equal and zero otherwise.

[14] This would mean that if the instruction *after the next one* attempts to use the value just loaded, the pipeline would stall for 1 clock cycle.

Note that certain pre-defined special conditionals exist and can be used in various expressions (mainly to determine side effects of operations). ISDL predefines the following special conditionals:[11]

- `OVF`: This behaves like a flag that is set if the action of an operation resulted in an overflow condition. This flag is only available for RTL descriptions of side effect actions.
- `CHANGED(x)`: This is a function that returns true only if the value of the storage represented by storage reference `x` changed as a result of the action of an operation. This conditional is only available in the side effects action definition of an operation description.
- `LIMIT`: This flag is set if limiting of the result took place as a result of an operation action. It is also set if limiting took place as a result of a simple move (i.e. an RTL assignment). This conditional is only available in the side effects action definition of an operation.

6.1.4 RTL Action

The RTL action description is used in operation definitions to describe the desired effect of the operation on the machine state. It is therefore based on assignments that describe the changes in values of various portions of the machine state. Syntactically, this description consists of a sequence of one or more RTL statements. The following is an example taken from the SPAM VLIW description:

```
Control_jumpcz RI       { Control.OP = 0x4 ;
                          Control.RI = RI ; }
                        { if (RI == 0) { PC <- JR ; }; }
                        {}
                        { Cycle=1; Size=1; }
                        { Latency=1; }
```

This example defines a conditional jump operation. The jump will be executed if a given register is zero, and will be ignored otherwise. The RTL action described in the operation first performs a test to see if the given register is zero.[12] If so, it then loads the Program Counter from `JR` (the jump register). Otherwise, it does nothing.[13].

6.1.5 Side Effects

The side effects description of an operation is described in the same way as the RTL action. In fact, logistically they are exactly the same: they both describe the effects of the operation on the state of the machine. However, it is important to realize that in ISDL, side effects are treated as if they take place **after** the RTL actions. Thus, the results of the RTL actions can be used in conditionals to determine what side effects will take place. Also note that the values of the special conditionals are also only valid after the RTL actions have taken place and can only be used in the side effects descriptions. There are also some other conceptual differences between side effects and actions which are investigated in Section 6.2.1.

The following is a simple example taken from the Motorola 56000 description:

```
DBM_Move LAGREGS, LAGREGS2  { DBM.OP = 0x20 ...; }
        { LAGREGS <- LAGREGS2; }
        { if (PLIMIT) { L_F = 1; }; }
        { Cycle = 0; Size = 0; }
        { Latency=1; }
```

[11] Additional conditionals may be defined by the tools that process ISDL - consult the tool documentation for more details.

[12] `RI` is actually a non-terminal action value but it is defined so that it returns the actual register.

[13] The Program Counter will be incremented automatically

On top of assignments and expressions, the ISDL version of RTL allows the use of the following flow control structures:

- `if` conditionals: These have one of two possible syntaxes:
  - `if (<expression>) {<statement>};`

    The statement can be **any** RTL statement (including another conditional) or group of such statements. This conditional evaluates *expression* first and if it evaluates to a non-zero value it executes *statement* else it does nothing.
  - `if (<expression>) {<statement1>} else {<statement2>};`

    This conditional evaluates *expression* first and if it evaluates to a non-zero value it executes *statement1* else it executes *statement2*. Both *statement1* and *statement2* can be **any** RTL statement or group of statements.
- `while` loops: The syntax for `while` loops is as follows:

  ```
  while (<expression>) {<statement>};
  ```

  This construct evaluates *expression* first and if non-zero it executes *statement*. Then it re-evaluates *expression* and if non-zero re-executes *statement* and so on. Note that *statement* can be **any** RTL statement or group of statements. Once *expression* evaluates to zero, execution proceeds to the next statement after the `while` (if any).
- `for` loops: The syntax for `for` loops is as follows:

  ```
  for (<stmnt1>;<expr2>;<stmnt3>) {<statement>};
  ```

  This construct first executes *stmnt1*. Then it evaluates *expr2* and if it evaluates to non-zero it executes *statement*. If *expr2* evaluates to zero, it proceeds with the statement after the `for`. If *expr2* evaluated to non-zero, then after *statement* has been executed, it executes *stmnt3*. It then re-evaluates *expr2* and if non-zero it executes *statement* and so on. Note that *statement* can be **any** RTL statement or group of statements. Usually, `expr2` is a conditional.
- `switch` statements: `switch` statements have the following syntax:

  ```
  switch (<expression>) {
      case <int1>: {<statement1>};
      case <int2>: {<statement2>};
      ...
      default: {<statementd>};
  };
  ```

  This construct fist evaluates *expression* into a bit pattern. Then it compares this bit pattern with *int1* which is a constant integer. If they are the same it executes *statement1* and proceeds to the statement after the switch.[10]. If they do not match, it tries to match the bit pattern with *int2* and if successful executes *statement2* and so on until a match is found or the default case is reached. If the default case is reached, *statementd* is executed and execution proceeds to the statement after the `switch`. Note that the statements can be **any** RTL statements or group of statements. Also note that the default case is optional in which case if none of the cases matched, none of the statements get executed and execution proceeds with the statement after the `switch`.

[10]Note that this is different from languages like C, in which an explicit `break` statement must be used to prevent execution from falling through to the next `case` statement

It assumes inputs and outputs of width `w`. Arguments `x` and `y` must be expressions of width at least `w`.

- `NOT(x,w)`: The bitwise "NOT" function. Each bit in the result is set if the corresponding bit in the input argument was cleared and vice versa. It assumes that the input and output are of width `w`. Argument `x` must be an expression of width at least `w`.
- `ASL(x,y,c,f,w)`: The shift left function. This shifts argument `x` as many positions to the left as denoted by the value of argument `y` interpreted as an integer. The value of flag `c` is shifted in from the right. The first bit to the left is stored in flag `f`. Argument `x` must be an expression of width at least `w`. Argument `y` must be an expression. Argument `c` must be an expression of width 1. Argument `f` must be a storage reference of width 1.
- `ASR(x,y,c,f,w)`: The shift right function. This shifts argument `x` as many positions to the right as denoted by the value of argument `y` interpreted as an integer. The value of flag `c` is shifted in from the left. The first bit to the right is stored in flag `f`. Argument `x` must be an expression of width at least `w`. Argument `y` must be an expression. Argument `c` must be an expression of width 1. Argument `f` must be a storage reference of width 1.
- `RND(x,w,m)`: The rounding function. This will round the value of argument `x` interpreted as an integer, to `w` bits. Argument `m` is a constant string representing the mode to be used. ISDL currently recognizes the modes "`up`", "`down`", "`near`", and "`equ`". Modes "`up`" and "`down`" round up (ceiling) and down (floor) respectively. Mode "`near`" rounds to the nearest integer, while mode "`equ`" rounds to the nearest integer but without the bias built into mode "`near`" because of the asymmetric positive and negative ranges of the values that can be represented. Argument `x` must be an expression of width at least `w`.
- `ABS(x,w)`: The absolute value function. It takes an integer `x` as input (in two's complement notation), and outputs `x` if `x` is positive or the negation of `x` if `x` is negative. It assumes that both input and output are of width at least `w` (in bits). The argument `x` must be an expression of width `w`. This operator will set the special conditional `OVF` if there is an overflow condition.[9] This operator may also set the `LIMIT` special conditional if limiting of the result took place.
- `EVAL(a)`: The instruction evaluation function. It takes as input the address `a` of an instruction to execute and executes it. This is identical to a normal instruction fetch and execution with only one exception: it does not auto-increment the Program Counter. The input parameter `a` must be an integer representing a valid instruction address.
- `HALT()`: The processor halt function. This is a marker function indicating that the processor stops execution until an external interrrupt occurs. It returns the value `NULL`.
- `NOP()`: The "no operation" function. This is a marker function indicating that the functional unit does not perform any operation. It returns the value `NULL`.
- `NULLOP()`: The NULL operation function. This is a marker function indicating that this is not a real operation but rather one added for syntactic convenience in the assembly. Operations which include the `NULLOP` function in their action RTL should contain *only* the `NULLOP` function in the action RTL and should contain a blank side-effects statement. They should also contain no bitfield assignments or costs and timing clauses. The `NULLOP` returns the value `NULL`.

Further functions may be defined and added to the individual tools that process ISDL depending on the implementation of these tools.

- Any expression enclosed within parentheses `()`. This has the same value as the expression itself; parentheses are used to show precedence.

[9] An overflow condition may arise because the negative range in two's complement is bigger than the positive range by one. Therefore, when negating the most negative number that can be represented, the result overflows.

the type of overflow arithmetic to be used. It can be either "`sat`" denoting saturation arithmetic or "`trn`" denoting truncation arithmetic. Arguments `x` and `y` must be expressions of width sufficient to hold the given format. If the storage is wider than necessary the numbers will be assumed to be aligned to the least significant bit. This function will set the special conditional `OVF` if an underflow condition occurs. *The current version of ISDL only supports IEEE format floating point representations.*

- `FMUL(x,y,e,m,t)`: The floating-point multiply function. This multiplies argument `x` with argument `y` interpreting them both as floating-point values. It assumes a format consisting of `m` mantissa bits and `e` exponent bits. The argument `t` is a string denoting the type of overflow arithmetic to be used. It can be either "`sat`" denoting saturation arithmetic or "`trn`" denoting truncation arithmetic. Arguments `x` and `y` must be expressions of width sufficient to hold the given format. If the storage is wider than necessary the numbers will be assumed to be aligned to the least significant bit. This function will set the special conditional `OVF` if an overflow condition occurs. *The current version of ISDL only supports IEEE format floating point representations.*
- `FDIV(x,y,e,m,t)`: The floating-point divide function. This divides argument `x` by argument `y` interpreting them both as floating-point values. It assumes a format consisting of `m` mantissa bits and `e` exponent bits. The argument `t` is a string denoting the type of overflow arithmetic to be used. It can be either "`sat`" denoting saturation arithmetic or "`trn`" denoting truncation arithmetic. Arguments `x` and `y` must be expressions of width sufficient to hold the given format. If the storage is wider than necessary the numbers will be assumed to be aligned to the least significant bit. This function will set the special conditional `OVF` if an underflow condition occurs. *The current version of ISDL only supports IEEE format floating point representations.*
- `FEQ(x,y,e,m)`: The floating point equality comparison function. Its value is 1 if and only if argument `x` and argument `y` represent the same floating point number, and 0 otherwise. It assumes a format consisting of `m` mantissa bits and `e` exponent bits. Arguments `x` and `y` must be expressions of width sufficient to hold the given format. If the storage is wider than necessary the numbers will be assumed to be aligned to the least significant bit. *The current version of ISDL only supports IEEE format floating point representations.*
- `FLT(x,y,e,m)`: The floating point "less than" comparison function. Its value is 1 if and only if argument `x` represents a floating point number less than the one argument `y` represents. It assumes a format consisting of `m` mantissa bits and `e` exponent bits. Arguments `x` and `y` must be expressions of width sufficient to hold the given format. If the storage is wider than necessary the numbers will be assumed to be aligned to the least significant bit. *The current version of ISDL only supports IEEE format floating point representations.*
- `FGT(x,y,e,m)`: The floating point "greater than" comparison function. Its value is 1 if and only if argument `x` represents a floating point number greater than the one argument `y` represents. It assumes a format consisting of `m` mantissa bits and `e` exponent bits. Arguments `x` and `y` must be expressions of width sufficient to hold the given format. If the storage is wider than necessary the numbers will be assumed to be aligned to the least significant bit. *The current version of ISDL only supports IEEE format floating point representations.*
- `AND(x,y,w)`: The bitwise "AND" function. Each bit in the result is set if the corresponding bits in both input arguments were set, and cleared otherwise. It assumes inputs and outputs of width `w`. Arguments `x` and `y` must be expressions of width at least `w`.
- `OR(x,y,w)`: The bitwise "OR" function. Each bit in the result is set if the corresponding bit in either input argument was set, and cleared otherwise. It assumes inputs and outputs of width `w`. Arguments `x` and `y` must be expressions of width at least `w`.
- `XOR(x,y,w)`: The bitwise "exclusive OR" function. Each bit in the result is set if one and only one of the corresponding bits in the input arguments was set, and cleared otherwise.

bits. The argument `t` is a string denoting the type of overflow arithmetic to be used. It can be either `sat` denoting saturation arithmetic or `trn` denoting truncation arithmetic. Arguments `x` and `y` must be expressions of width at least `w`. This function may set the special conditional `OVF` if an overflow or underflow condition occurs. This function may also set the `LIMIT` special conditional if limiting of the result took place.

- `MUL(x,y,wi,wo,t)`: The integer multiply function. This function multiplies argument `x` with argument `y`. It assumes inputs of width `wi` and outputs `wo` bits. It will mask out all irrelevant bits. The argument `t` is a string denoting the type of overflow arithmetic to be used. It can be either "`sat`" denoting saturation arithmetic or "`trn`" denoting truncation arithmetic. Arguments `x` and `y` must be expressions of width at least `wi`. This function may set the special conditional `OVF` if an overflow condition occurs. This function may also set the `LIMIT` special conditional if limiting of the result took place.
- `DIV(x,y,wx,wy,wo,t)`: The integer divide function. This function divides argument `x` by argument `y`. It assumes that `x` is of width `wx` and `y` is of width `wy`. It outputs `wo` bits. It will mask out all irrelevant bits. The argument `t` is a string denoting the type of overflow arithmetic to be used. It can be either "`sat`" denoting saturation arithmetic or "`trn`" denoting truncation arithmetic. Arguments `x` and `y` must be expressions of width at least `wx` and `wy` respectively.
- `MOD(x,y,wx,wy,wo,t)`: The integer modulo function. This function divides argument `x` by argument `y` and returns the remainder. It assumes that `x` is of width `wx` and `y` is of width `wy`. It outputs `wo` bits. It will mask out all irrelevant bits. The argument `t` is a string denoting the type of overflow arithmetic to be used. It can be either "`sat`" denoting saturation arithmetic or "`trn`" denoting truncation arithmetic. Arguments `x` and `y` must be expressions of single register type and width at least `wx` and `wy` respectively.
- `FTOI(x,w,e,m,t)`: The cast function to convert floating point numbers to integers. The argument `x` is interpreted as a floating point number with `e` bits of exponent and `m` bits of mantissa. The function returns a `w`-bit wide, two's complement representation of `x`. The argument `t` is a string denoting the type of overflow arithmetic to be used. It can be either "`sat`" denoting saturation arithmetic or "`trn`" denoting truncation arithmetic in case `x` is too large to be represented in `w` bits. Argument `x` must be an expression of width sufficient to hold the given format. If the storage is wider than necessary the numbers will be assumed to be aligned to the least significant bit. *The current version of ISDL only supports IEEE format floating point representations.*
- `ITOF(x,w,e,m,t)`: The cast function to convert integer numbers to floating point. The argument `x` is interpreted as a two's complement integer `w` bits wide. The function returns a floating point number `e` + `m` bits in length with an `e`-bit exponent and an `m`-bit mantissa.The argument `t` is a string denoting the type of overflow arithmetic to be used. It can be either "`sat`" denoting saturation arithmetic or "`trn`" denoting truncation arithmetic in case `x` is too large to be represented in `e` + `m` bits. *The current version of ISDL only supports IEEE format floating point representations.*
- `FADD(x,y,e,m,t)`: The floating-point addition function. This adds argument `x` to argument `y` interpreting them both as floating-point values. It assumes a format consisting of `m` mantissa bits and `e` exponent bits. The argument `t` is a string denoting the type of overflow arithmetic to be used. It can be either "`sat`" denoting saturation arithmetic or "`trn`" denoting truncation arithmetic. Arguments `x` and `y` must be expressions of width sufficient to hold the given format. If the storage is wider than necessary the numbers will be assumed to be aligned to the least significant bit. This function will set the special conditional `OVF` if an overflow condition occurs. *The current version of ISDL only supports IEEE format floating point representations.*
- `FSUB(x,y,e,m,t)`: The floating-point subtraction function. This subtracts argument `y` from argument `x` interpreting them both as floating-point values. It assumes a format consisting of `m` mantissa bits and `e` exponent bits. The argument `t` is a string denoting

- "<": The logical less-than operator. The resulting expression is one if the first input expression is less than the second when both input expressions are interpreted as two's complement integers, and zero otherwise. The input expressions must have the same width.
- ">=": The logical greater-than-or-equal operator. The resulting expression is one if the first input expression is greater than or equal to the second when both input expressions are interpreted as two's complement integers, and zero otherwise. The input expressions must have the same width.
- "<=": The logical less-than-or-equal operator. The resulting expression is one if the first input expression is less than or equal to the second when both input expressions are interpreted as two's complement integers, and zero otherwise. The input expressions must have the same width.

- A function operator. This is a name followed by a comma-separated list of parameters enclosed in parentheses "()". The parameters can be strings, constant numbers or floats, constant characters and expressions. The resulting value is defined by the function and the parameters. The ISDL version of RTL predefines the following functions:
  - `EXT(v,c,e)` This is the extension function. It takes the last `c` bits of the expression `v` and extends it to an `e`-bit bitfield by inserting zeros in the MSB. It can both extend and contract the representation of `v`. In other words, `e` can be smaller *or* larger than `c`.
  - `SEXT(v,c,e)` This is the sign extension function. It takes the last `c` bits of the expression `v` and sign extends it to the same value represented as an `e`-bit two's complement integer. It can both extend and contract the representation of `v` (which is interpreted as a two's complement integer). In other words, `e` can be smaller *or* larger than `c`.
  - `ADDC(x,y,c,f,w,t)`: The integer add with carry function. This function adds argument `x` to argument `y` and the input carry bit `c`. The bit `f` is set to the value of the produced carry. The function assumes inputs and output of width `w`. It will mask out all irrelevant bits. The argument `t` is a string denoting the type of overflow arithmetic to be used. It can be either "`sat`" denoting saturation arithmetic or "`trn`" denoting truncation arithmetic. Arguments `x` and `y` must be expressions of width at least `w`. Argument `c` must be an expression of width 1. Argument `f` must be a storage reference of width 1. This function will set the special conditional `OVF` if there was an overflow. This function may also set the `LIMIT` special conditional if limiting of the result took place.
  - `SUBC(x,y,c,f,w,t)`: The integer subtract with borrow function. This function subtracts argument `y` from argument `x` and the input borrow bit `c`. The bit `f` is set to the value of the produced borrow. The function assumes inputs and output of width `w`. It will mask out all irrelevant bits. The argument `t` is a string denoting the type of overflow arithmetic to be used. It can be either "`sat`" denoting saturation arithmetic or "`trn`" denoting truncation arithmetic. Arguments `x` and `y` must be expressions of width at least `w`. Argument `c` must be an expression of width 1. Argument `f` must be a storage reference of width 1. This function will set the special conditional `OVF` if there was an overflow or underflow. This function may also set the `LIMIT` special conditional if limiting of the result took place.
  - `ADD(x,y,w,t)`: The integer add function. This function adds argument `x` to argument `y`. It assumes inputs and output of width `w`. It will mask out all irrelevant bits. The argument `t` is a string denoting the type of overflow arithmetic to be used. It can be either "`sat`" denoting saturation arithmetic or "`trn`" denoting truncation arithmetic. Arguments `x` and `y` must be expressions of width at least `w`. This function may set the special conditional `OVF` if an overflow condition occurs. This function may also set the `LIMIT` special conditional if limiting of the result took place.
  - `SUB(x,y,w,t)`: The integer subtract function. This function subtracts argument `y` from argument `x`. It assumes inputs and output of width `w`. It will mask out all irrelevant

- "+": The arithmetic addition operator. The result of the expression is the arithmetic sum of the two input expressions interpreted as two's complement integers. The sizes of the input expressions must match that of the output expression. This operator may set the special conditional `OVF` if an overflow condition occurs. This operator may also set the `LIMIT` special conditional if limiting of the result took place.
- "-": The arithmetic subtraction operator. The result of the expression is the arithmetic subtraction of the second input expression from the first, given that both input expressions are interpreted as two's complement integers. The sizes of the input expressions must match that of the output expression. This operator may set the special conditional `OVF` if an overflow or underflow condition occurs. This operator may also set the `LIMIT` special conditional if limiting of the result took place.
- "*": The arithmetic multiplication operator. The result of the expression is the arithmetic product of the two input expressions interpreted as two's complement integers. The sizes of the input expressions must match that of the output expression. This operator may set the special conditional `OVF` if an overflow condition occurs. This operator may also set the `LIMIT` special conditional if limiting of the result took place.
- "/": The arithmetic division operator. The result of the expression is the integer portion of the result obtained from dividing the first input expression by the second input expression, given that both input expressions are interpreted as two's complement integers. The sizes of the input expressions must match that of the output expression.
- "%": The arithmetic modulo operator. The result of the expression is the remainder portion of the result obtained from dividing the first input expression by the second input expression, given that both input expressions are interpreted as two's complement integers. The sizes of the input expressions must match that of the output expression.
- "|": The bitwise "OR" operator. Each bit in the resulting expression is set if either of the corresponding bits of the input expressions was set, and cleared otherwise. The sizes of the input expressions must match that of the output expression.
- "&": The bitwise "AND" operator. Each bit in the resulting expression is set if both of the corresponding bits of the input expressions were set, and cleared otherwise. The sizes of the input expressions must match that of the output expression.
- "^": The bitwise "XOR" operator. Each bit in the resulting expression is set if exactly one of the corresponding bits of the input expressions was set, and cleared otherwise. The sizes of the input expressions must match that of the output expression.
- ">>": The logical shift right operator: The resulting expression is the bit pattern resulting from shifting the first input expression right the same number of bits as the second input expression interpreted as a simple integer. The corresponding number of zeros are inserted from the left. The sizes of the input expressions must match the size of the output expression.
- "<<": The logical shift left operator: The resulting expression is the bit pattern resulting from shifting the first input expression left the same number of bits as the second input expression interpreted as a simple integer. The corresponding number of zeros are inserted from the right. The sizes of the input expressions must match the size of the output expression.
- "==": The logical equation operator. The resulting expression is one if the two input expressions have identical bit-patterns and zero otherwise. The input expressions must have the same width.
- ">": The logical greater-than operator. The resulting expression is one if the first input expression is greater than the second when both input expressions are interpreted as two's complement integers, and zero otherwise. The input expressions must have the same width.

reference, followed by the "`<-`" symbol, followed by an RTL expression, and terminated by a "`;`". This kind of statement has the effect of assigning the value of the expression to the state (or portion thereof) represented by the storage reference. ISDL uses this construct to express the effect of operations on the machine state (the storage).

A temporary storage declaration consists of the keyword "`int`" followed by an integer declaring the width of the storage in bits, followed by the name to be used to refer to the storage. Its scope is the current RTL definition only. The following example is taken from the Motorola 56000 description:

```
int 16 TMP;
```

This declares a temporary register called `TMP` which can hold a 16-bit integer value.
The ISDL version of RTL defines the following options for expressions:

- A constant: This can be either an integer in decimal or hexadecimal notation, or a floating point number. The value of this is a 32-bit representation of the constant if it is an integer or an IEEE floating point representation if it is a floating point number.
- A storage reference: The value of this expression depends on whether it is being used in a calculation or it is being assigned to. If being used in the calculation, the value of this expression is the contents of the location being referred to. If being assigned to, this expression refers to the actual location. Storage references in RTL must be of the single register type.
- A temporary register reference. This acts just like a storage reference except that it refers to temporary storage instead of the processor state.
- A token name: The value of this expression is the return value of the token (see Section 4.1.1).
- A non-terminal name: The value of this expression is the value of the expression given as an RTL Action or Side Effect value in the non-terminal definition (see Section 4.2.1). Which one should be used depends on the context of the current RTL expression. If the current RTL expression is within the RTL Action definition of either an operation definition or a non-terminal definition then we use the non-terminal RTL Action expression. If the current RTL expression is within the Side Effects definition of either an operation definition or a non-terminal definition then we use the Side Effects value of the non-terminal.
- A unary operator followed by an expression. The ISDL version of RTL currently accepts the following unary operators:
  - "`~`": The bitwise "NOT" operator. Each bit in the resulting expression is set if the corresponding bit in the input expression was cleared, and cleared otherwise. If necessary the input expression is truncated or extended on the left with zeros to match the size of the output expression.
  - "`!`": The logical "NOT" operator. The resulting expression is the integer representation of 1 if the input expression is zero, and zero otherwise.
  - "`-`": The arithmetic negation operator. The resulting value is the two's complement negation of the input value interpreted as a two's complement integer. The size of the resulting value is the same as the size of the input value - if necessary the input value should be sign extended using the function operator `SEXT()`. This operator will set the special conditional `OVF` if there is an overflow condition.[8] This operator may also set the `LIMIT` special conditional if limiting of the result took place.
- An expression, followed by a binary operator, followed by another expression. The ISDL version of RTL currently accepts the following binary operators:

[8] An overflow condition may arise because the negative range in two's complement is bigger than the positive range by one. Therefore, when negating the most negative number that can be represented, the result overflows.

- "<<": The logical shift left operator. The resulting expression is the same as the binary representation of the first given expression, shifted left by the number of bits corresponding to the integer value of the second given expression. Zeros are inserted as appropriate on the right side. For example, the value of the expression `0xfa5 << 0x4` is `0xfa50`.

- An expression within parentheses "()": Parentheses are used to denote precedence where necessary.

The above covers the bitfield assignments for the current instruction word. However, it is possible for instructions to need additional words for large constants (e.g. addresses for the targets of jumps or large constant values to be transferred to registers). In this case, a special form of bitfield assignment must be used (in addition to the usual bitfield assignments). The syntax of this form of assignment is:

```
Additional(<int>, <bitfield assignments>);
```

The word `Additional` is a keyword and may not be used anywhere else in an ISDL description. The integer provided is the identifier of the additional word being set. ISDL allows each instruction to access multiple additional words. The integer defines which one of the multiple words is being set. Word indices begin with 0. The bitfield assignment is a normal bitfield assignment, except that it sets the additional word instead of the current instruction word. Any additional words defined for an instruction are added immediately following that instruction in the instruction stream. Below is an example taken from the Motorola 56000 description:

```
Main_JCLR IMM, MEMS, ADRM, ADDRESS
    { Main.OP = 0x80 | (IMM & 0x1F) |
      (MEMS << 6); DBM.OP = 0x0A;
      DBM.MODE = 0x40 | ADRM;
      Additional(0, Split.ADDR = ADDRESS;); }
...
```

In this example, the current instruction needs an additional word to store the address that forms the destination of the jump. The value to be stored in this address is returned by the non-terminal `ADDRESS` and the bitfield assignment that stores it in the additional word is the one contained in the last line of the above example. Note that it stores it at the additional word with index 0 which immediately follows the current instruction word.

There are two additional notes concerning bitfield assignments. First of all, all bitfields are initialized with a value of 0 before any bitfield assignments take place. So if no operation sets a particular subfield, you are guaranteed that its value is 0. Secondly, the assignments happen in the order the fields are given. In other words, if you have an instruction consisting of operations from five fields, you are guaranteed that the assignments of the operation from the first field will happen first, those of the operation from the second field will happen next, and so on.

6.1.3 Register Transfer Language Syntax

Before we dive into the description of RTL actions and Side Effect clauses, it might be beneficial to examine in detail the language that is used to express both, the Register Transfer Language. While RTL is a simple language, a proper understanding of it is necessary in order to allow one to understand how it is used in forming the actions and side effect clauses necessary for operation definitions. Furthermore, ISDL admits an extended (and extensible) version of RTL that does not specify the operations themselves and therefore new types of operations can be added by reconfiguring the tools that use ISDL as input. This means that an understanding of the *structure* of RTL is just as important as the understanding of the language itself.

RTL is an expression based language. The basic entity in the subset of RTL that ISDL accepts, is a statement. A statement can be a control structure, an assignment, a temporary storage declaration, a token name, an RTL function call, or a list of statements. An assignment consists of a storage

6.1.2 Bitfield Assignment

The bitfield assignment clause consists of a number of C-like assignments that set the instruction word subfields (as defined in the Format section of the description) to the appropriate binary images. Images can be treated as integers of the corresponding width. An assignment consists of the name of the subfield or of a split function (see Section 4.3), followed by an "`=`" sign, followed by an expression whose value is the integer to be assigned to the bitfield. An expression can be one of:

- A constant integer.
- The name of a token or a non-terminal that appears in the parameter list of the operation. The value of this expression is the integer represented by the return value of the token or non-terminal.
- The name of a subfield. The value of this expression is the binary value that has been assigned to that subfield by any previous bitfield assignments, or zero if it has never been assigned in the current instruction. Bitfield assignments for operations happen in the order the fields are declared. Bitfield assignments in non-terminals used by an operation happen before the bitfield assignments of the operation itself.
- The special keyword `CURRENT`. The value of this expression is the integer representing the address of the current instruction in the instruction memory.
- A unary operator (such as "`~`") followed by any expression. ISDL currently only recognizes the "`~`" unary operator which inverts the bits in the binary representation of the expression.
- An expression followed by a binary operator followed by another expression. ISDL currently recognizes the following binary operators:
  - "`&`": The bitwise "AND" operator. The resulting expression has a binary representation in which a bit is set if both the corresponding bits of the input expressions were set. The bit is cleared otherwise. For example, the value of the expression `0xfa5 & 0x23` is `0x21`.
  - "`|`": The bitwise "OR" operator. The resulting expression has a binary representation in which a bit is cleared if both the corresponding bits of the given expressions were cleared. The bit is set otherwise. For example, the value of the expression `0xfa5 | 0x23` is `0xfa7`.
  - "`^`": The bitwise "XOR" operator. The resulting expression has a binary representation in which a bit is set if one of the corresponding bits of the given expressions was set but not both. The bit is cleared otherwise. For example, the value of the expression `0xfa5 ^ 0x23` is `0xf87`.
  - "`+`": The arithmetic addition operator. The resulting operation is the arithmetic sum of the integers representing each of the given expressions. For example, the value of the expression `0xfa5 + 0x23` is `0xfc8`.
  - "`-`": The arithmetic subtraction operator. The resulting operation is the value of the integer representing the left-hand expression minus the value of the integer representing the right-hand expression. For example, the value of the expression `0xfa5 - 0x23` is `0xf82`. If the result is less than zero the resulting value is undefined.
  - "`*`": The arithmetic multiply operator. The resulting operation is the arithmetic multiple of the integers representing each of the given expressions. For example, the value of the expression `0xfa5 * 0x23` is `0x2238f`.
  - "`>>`": The logical shift right operator. The resulting expression is the same as the binary representation of the first given expression, shifted right by the number of bits corresponding to the integer value of the second given expression. Zeros are inserted as appropriate on the left side. For example, the value of the expression `0xfa5 >> 0x4` is `0xfa`.

6 Instruction Set

The Instruction Set section describes each operation available in the instruction set using the information defined in the previous sections. This section has the most complicated syntax and is usually the longest part of an ISDL description.

The contents of this section consist of one or more *Field* definitions. A field is a virtual part of the instruction word. It generally corresponds to a single functional unit in a VLIW architecture that can be controlled independently and can execute one of many mutually exclusive operations. Fields can theoretically be set independently of each other and represent operations that can be performed in parallel; one on each of the functional units of a VLIW architecture. In practice, this is not always possible (see Section 7). If the architecture has no instruction level parallelism then only one field will be available in the Instruction Set section.

6.1 Instruction Set Syntax

The Instruction Set section consists of one or more *Field* definitions. Each field definition consists of:

- The keyword `Field`.
- A name for the field (which is used in various other sections of the description), followed by a ":".
- A list of one or more operation definitions.

A field definition is considered to have ended when the next field definition starts or the next section starts.

An operation definition defines a single operation in the instruction set. It consists of six parts:

- An assembly syntax definition. This serves a dual purpose. It first names the operation so that it can be referred to and identified in later portions of the description. Even more importantly, it defines the exact assembly syntax that the operation should respond to.
- A bitfield assignment clause. This is an expression which assigns the appropriate binary image for the operation to the appropriate subfields of the instruction word. It is used to generate the assembler and disassembler (including the disassembler built into the simulator).
- A Register Transfer Language (RTL) action that describes the effect of the operation on the processor state.
- An RTL Side Effects clause. This is described in the same form as the RTL action above but describes any side effects the instruction might have, instead of its actual intended purpose.
- A cost expression clause. This is a set of one or more expressions that determine the various costs of the operation (such as code size, cycle cost, etc.)
- A timing expression clause. This is a set of one or more expressions that determine the timing parameters of the operation (such as latency and usage).

Each part of an operation definition has its own syntax.

6.1.1 Assembly Syntax Definition

The assembly syntax definition is relatively simple. It consists of an operation name and a comma-separated list of parameters. The parameters are the names of tokens and non-terminals. These tokens and/or non-terminals must be defined in the Global Definitions section of the description.

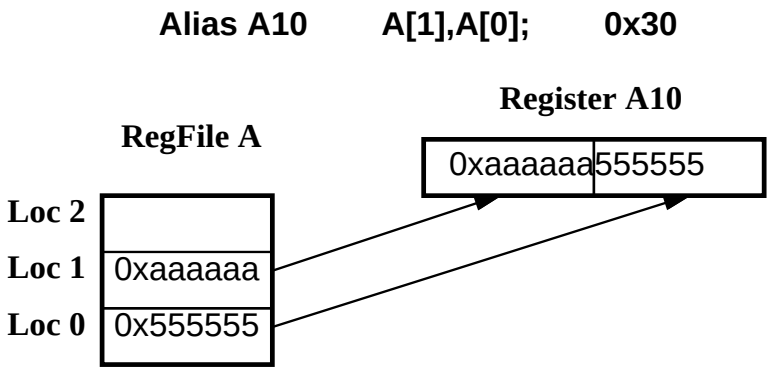

Figure 7: **The relation between existing Storage and an alias.**

- `MMIO`: This represents Memory Mapped I/O ports. Zero or more ports may be defined in a description. They behave like addressed storage units with one exception: reading from a port location will not necessarily return the value that was last written in the port location. Also writing to a location may have side effects in hardware. In that respect they behave very much like a group of control registers.
- `ProgramCounter`: This is a special control register. It must be explicitly declared.
- `Stack`: This is an addressed storage unit but with the special restriction that its address during operation processing comes from an already declared `Register`. In the example above this register is the `SP`. This register must follow the name of the stack unit in the stack definition, placed within `()`.

5.3 Alias Definitions

Below is an example of some alias definitions taken from the Motorola 56000 description:

```
Alias AREG      A[2][0x0-0x7],A[1],A[0];        0x38
Alias SSH       SS[SP][0x10-0x1f];              0x10
Alias A10       A[1],A[0];                      0x30
Alias LMEM      XMEM[0-0xFFF],YMEM[0-0xFFF];    0x1000 , 0x30
```

5.3.1 Meaning Of Alias Definitions

The syntax of Alias definitions has already been defined in Section 5.1. The meaning of an alias definition is the same as that of a storage definition. It declares a storage unit which may be a single register or an addressed unit. The only real difference is that it is made up of portions of storage units that were already defined. This means that writing to the unit corresponding to an alias will affect the contents of other storage units and vice versa. Let's illustrate this with an example: in the example above we have the alias:

```
Alias A10       A[1],A[0];                      0x30
```

Note that `A` is the name of a register file unit with 3 registers of 24 bits each. This alias defines another single register storage unit 48 bits wide by concatenating in order `A[1]` and `A[0]`. This makes `A[1]` the most significant word. Assume that the contents of `A[1]` are 0xaaaaaa and the contents of `A[0]` are 0x555555. Then the contents of the single register `A10` are 0xaaaaaa555555. Figure 7 shows this relation graphically. Suppose now that we write the value 0x101010 in `A[0]`. All of a sudden, the value of `A10` changes to 0xaaaaaa101010 even though we never explicitly wrote to `A10`. Similarly, if we were to write the value 0xaaaaaa222222 to `A10`, the value of `A[0]` would become 0x222222 even though we never wrote a new value into `A[0]`. Even though this feature is complicated and confusing, it is nevertheless necessary since real architectures (such as the Motorola 56000) do in fact have operations that make use of it.

Aliases can also emulate addressed storage units - just combine addressed units in parallel. For example:

```
Alias LMEM      XMEM[0-0xFFF],YMEM[0-0xFFF];    0x1000,0x30
```

In the above example, two 24-bit memories are concatenated to form one 48-bit memory. The depth of all 3 memories is 4096 locations. When emulating addressed storage units by concatenating units in an alias, make sure that the address ranges given match - an error results if they do not.

Also note that alias definitions can use *portions* of existing storage:

```
Alias AREG      A[2][0x0-0x7],A[1],A[0];        0x38
```

In this example, note that we only use the 8 least significant bits of `A[2]`. Similarly we could have used portions of other pieces of storage too. This is applicable not only to the emulation of single register units but also addressed units as well.

5. A subset of complete locations in an addressed unit: This consists of the name of the addressed unit followed by a range operator. This reference has the same type as an addressed storage unit. Example: `XMEM[0-15]`.

6. A subset of partial locations in an addressed unit: This consists of the name of the addressed unit followed by two range operators. This reference has the same type as an addressed storage unit. Example: `XMEM[0-15][0-7]`.

Also note that in ISDL's simplified version of RTL, you can use the special storage reference `NULL` which can be written to without affecting machine state, and when read from returns an unspecified result. This reference has the same type as a single register.

5.2 Storage Definitions

Below are some examples of storage definitions (taken from the Motorola 56000 description):

```
Section Storage

Instruction Memory IM    = 0x1000 , 0x18
Memory  XMEM             = 0x1000 , 0x18
RegFile AGU_R            = 0x8 , 0x18
RegFile A                = 0x3 , 0x18
Register LA              = 0x10
MMIO XPort               = 0x40 , 0x18
CRegister LC             = 0x10
ProgramCounter PC        = 0x10
Stack SS(SP)             = 0xf , 0x20
```

5.2.1 Storage Unit Types

The current version of ISDL defines the following types of storage units:

- `Memory`: This is an addressed storage type used mainly to declare memories. Multiple memory units of different sizes can be defined in the same description. The memory unit from which instructions are loaded has to be identified during its declaration by the use of the keyword `Instruction`. Because of this restriction, at least one memory must be defined for each architecture.

- `RegFile`: This is also an addressed storage type and is commonly used to declare register files closely associated with the functional units of the processor. Zero or more register files may be defined. Note that even though Register Files and Memories behave identically in an ISDL description (with the only exception being the declaration of an Instruction Memory), various tools that process ISDL descriptions (such as the simulator generators and the code-generator generators) might depend on the distinction between the two.

- `Register`: This represents a single register unit specifically devoted to data processing (for control registers see below). It can be thought of as a register file with only a single location in it.

- `CRegister`: This represents control registers. Like normal registers, these are single register units but are devoted to controlling various aspects of the hardware. Because of this, the value read from a control register does not necessarily equal the value last written into the same register. Also, writing into control registers might cause unexpected side effects (such as resetting interrupt modes etc.). The generated tools (and in particular the code generators) will take this into account.

5 Storage

The Storage section of an ISDL description defines and names all storage resources visible to the programmer.

5.1 Storage Definition Syntax

The Storage section of an ISDL description consists of a list of storage definitions or alias definitions. Storage definitions declare new storage units available in the processor being described. Aliases are used to rename already defined storage units or portions thereof.

Each storage definition consists of the following parts:

- *Type*: This declares what type of storage resource we are defining. It consists of one of the following keywords: `Instruction Memory`, `Memory`, `RegFile`, `Register`, `MMIO`, `CRegister`, `ProgramCounter`, `Stack`.
- *Name*: This defines the name that will be used to refer to the storage from now on.[7]
- An "`=`" sign to delimit the size declaration.
- *Size declaration*: There are two forms of size declarations: single register and addressed unit declarations. Single register declarations consist of a single integer representing the width in bits of the storage resource. Addressed unit declarations consists of two integers (representing the width in bits and depth in locations) separated by a "`,`".

Each Alias definition consists of the following parts:

- The keyword `Alias`.
- A name for the alias that can be used to refer to a particular part of the processor storage.
- A coma-separated list of one or more storage references terminated by a "`;`".
- A size declaration of the same type as the size declarations in the storage definitions.

The syntax of an alias definition involves *storage references*. Storage references have a number of varying forms depending on the form of the units they refer to (the examples correspond to the storage defined in Section 5.2):

1. Referring to a single and complete register of a single register type: the syntax for this reference is the register name. Example: `PC`.
2. Referring to a single and complete location in an addressed unit: The syntax for this is the name of the addressed unit followed by an index operator. An index operator consists of an integer within "`[ ]`". This reference has the same type as a single register. Example: `XMEM[12]`.
3. Referring to a single but partial register of a single register type: This consists of the name of the register followed by a range operator. A range operator consists of two integers within "`[ ]`" separated by a "`-`". This reference has the same type as a single register. Example: `LA[0-7]`.
4. Referring to a single but partial location in an addressed unit: This consists of the same type of reference as (2) above followed by a range operator. This reference has the same type as a single register. Example: `XMEM[12][0-7]`.

[7] The only exception to this general syntax is the definition for a stack: the name is followed by the name of the stack pointer (which must be the name of a register also defined for the architecture) in parentheses.

```
Split.DATA      DBM.OP+DBM.MODE+Main.OP;
```

This defines a function that will automatically take a binary image (of 24 bits in this case) and split it between the three 8-bit fields of the instruction word. We can then use the function Split.DATA in the operation definitions to assign large constants to the subfields.

The syntax of split function definitions begins with the keyword `Split`, followed by a dot, followed by the name of the function. Next comes a list of subfield names separated by a "`+`" and terminated by a semi-colon. For the purposes of forming the effective bitfield, the subfields are concatenated in left-to-right order with the MSB on the left. The effective field can be assigned with a command such as: `Split.Data = ...` in the bitfield assignment section of the operation definitions (see Section 6.1.2).

```
Non_Terminal      DXEA:
                '(' R_R ')' '+' N_R {$$ = 0x08 | (R_R & 0x7); } {AGU_R[R_R]}
                   { AGU_R[R_R] <- AGU_R[R_R] + AGU_N[N_R] } {} {} |
                '(' R_R ')' '-'     {$$ = 0x10 | (R_R & 0x7); } {AGU_R[R_R]}
                   { AGU_R[R_R] <- AGU_R[R_R] - 1 }   {} {} |
                '(' R_R ')' '+'     {$$ = 0x18 | (R_R & 0x7); } {AGU_R[R_R]}
                   { AGU_R[R_R] <- AGU_R[R_R] + 1 }
                       { Cycle = 2; Size = 0; } {} |
                '(' R_R ')'         {$$ = 0x00 | (R_R & 0x7); } {AGU_R[R_R]}
                   {} { Cycle = 2; Size = 0; } {} ;
```

The above example defines a non-terminal called **DXEA** which groups together four syntactic options:

- '(' R_R ')' '+' N_R (i.e. phrases of the type (*R_R* + *N_R*))
- '(' R_R ')' '-' (i.e. phrases of the type (*R_R*) -)
- '(' R_R ')' '+' (i.e. phrases of the type (*R_R*) +)
- '(' R_R ')' (i.e. phrases of the type (*R_R*))

where *R_R* and *N_R* represent the entities grouped together by the tokens R_R and N_R. Let's look at the syntax of the option definitions more closely. Each option consists of the syntax description listed above followed by an expression describing the return value, followed by the RTL Action Value, the RTL Side Effects value, the costs modifier and the timing modifier. For the sake of brevity we will only examine the syntax of the third option (probably the most interesting one).

The return value calculation is represented by the expression "{$$ = 0x18 | (R_R & 0x7); }". According to this, the last 3 bits of the return value are taken from the token R_R, the fourth and fifth bit are set and the rest are cleared. There are no bitfield assignments for this option (or any of the other options in this non-terminal).

The RTL Action value is "AGU_R[R_R]". This is a partial RTL action which represents a storage location (namely the register in the AGU register file that corresponds to the return value of R_R).

The Side Effect value is "AGU_R[R_R] <- AGU_R[R_R] + 1". This is a full RTL action describing an increment operation on the AGU register corresponding to the return value of the token R_R.

The Cost modifier is Cycle = 2; Size = 0;. This defines two cost values, Cycle and Size, and gives them the values of 2 and 0 respectively. These values will be added to the corresponding costs in the operation definition if this option is used in an instantiation. Since no Stall cost is defined, it's value is assumed to be 0.

The Timing modifier for this example is null. In this case, the timing parameters of the operation are not altered in any way if this option is used in an instantiation.

4.2.2 Other Non-Terminal Issues

Note that strings are not allowed in the syntax description of non-terminals. It is however entirely possible that a constant string will be necessary in the syntax description of a non-terminal. This problem can be solved by defining a new token for the constant string and using the token in the non-terminal definition. Also note the comments provided in Section 9.2.4.

4.3 Other Definitions

On top of token and non-terminal definitions, ISDL allows additional definitions to make descriptions shorter and more intuitive. The current version of ISDL allows the definition of split functions which make the task of creating binary images easier. These functions group together fields and subfields of the word format into larger entities to make it easier to assign large binary images to them. Below is an example taken from the Motorola 56000 description:

1. The binary image (the pattern of bits that need to be inserted into the instruction word to denote the operation). This image may depend on which non-terminal option is present in an operation. Non-terminals provide a return value to identify each option to the assembler so that it can generate the appropriate binary image. In fact, generally the return value will consist of the appropriate portion of the binary image. Non-terminals also allow bitfield assignments which can set the binary image directly.

2. The data manipulation action that the operation performs. An operation's action might vary depending on the actual parameters it receives. Given that the parameters themselves will vary depending on which option a particular instance of the non-terminal refers to, we need to identify the actual action performed. Within the non-terminal definition we list the action that corresponds to each particular option. This is the RTL action value. Typically we do not list a complete action but rather a portion of it which when combined with the RTL description of the operation definition results in a full description of what the operation action is.

3. Any side effects that the operation might have. The same comments that apply to operation actions apply to operation side effects as well. The relevant portion of the non-terminal definition is the RTL side effect value.

4. The cost of the operation. Each operation has one or more costs associated with it. These costs may vary according to the parameters of the operation and thus may vary according to which non-terminal option a particular instance refers to. Thus, each option within a non-terminal definition carries a cost modifier, which when combined with the cost function in the operation definition results in a full description of the cost for the operation.

5. The Timing of the operation. The same comments that apply to operation costs apply to operation timing as well. The effects of a non-terminal on timing are handled by listing a timing modifier for each option within a non-terminal definition.

Below is a description of the syntax of each of the six items in a non-terminal option definition:

1. The syntactic description of the option: This consists of a space separated list of characters, token names and non-terminal names in the order in which they should appear in the assembly.

2. The return value and possibly some bitfield assignments. This is intended to identify the option and possibly perform some of the bitfield assignments. The syntax of bitfield assignments is described in Section 6.1.2 so it is not described here. The syntax for the return value is the same as that of bitfield assignments except for the fact that the return value is assigned to the special variable `$$`. Effectively, the return value corresponds to whatever bitfield value was assigned to `$$`. The return value expression and bitfield assignments are enclosed in "{}".

3. The RTL Action Value: This is either a complete or partial RTL action of the same syntax as described in Section 6.1.3. It is enclosed in "{}".

4. The Side Effect value. This is either a complete or partial RTL side effect of the same syntax as described in Section 6.1.3. It is enclosed in "{}".

5. The Cost Description: This is a set of cost expressions of the same type as the cost expressions in the corresponding section of the operation definition. See Section 6.1.6 for details. It is enclosed in "{}".

6. The Timing Description. This is a set of timing parameters of the same type as the timing parameters in the corresponding section of the operation definition. See Section 6.1.7 for details. It is enclosed in "{}".

Below is a complete example of a token definition taken from the description of the Motorola 56000:

- The special token **NAME**. This token will be returned for any labels in the assembly code. Labels are symbolic names for instruction addresses. The syntax for declaring a label in assembly is:

  ```
  <label>: <instruction>
  ```

  The label can be used in any place where one would use the actual address of the assembly instruction associated with the label declaration. The return value of the token **NAME** is an integer representing the address of the instruction that the label stands for. It is therefore a good idea to accept the token **NAME** at any place where an instruction address could be used. The assemblers generated by ISDL are two-pass assemblers which means that the labels will be correct even if they are used before they are declared.

- Any operation name. All operation names are pre-defined as tokens by the tools processing ISDL.

Note that any syntactic entity other than the predefined tokens and characters *must* be declared explicitly as a token.

4.2 Non-Terminals

Just like tokens, Non-Terminals are intended to group syntactic elements to factor out common patterns in operation definitions. However, non-terminal definitions are much more flexible (and much more complex) than token definitions and can group together syntactically unrelated entities.

4.2.1 Non-Terminal Syntax

Non-Terminal definitions consist of three main parts:

1. The keyword **Non_Terminal**.
2. A name by which the non-terminal can be referred to in the definitions of other non-terminals or operations, followed by a colon.
3. a sequence of one or more options separated by the special character "|". This list is terminated by a semi-colon.

Each option itself consists of six items:

1. A list of characters, token names and/or non-terminal names in the order they should appear in the assembly syntax of the non-terminal being defined.
2. A return value and possibly bitfield assignments.
3. An RTL action value.
4. An RTL side effect value.
5. A cost modifier.
6. A timing modifier.

Since the different syntactic entities represented by a single token are all related, the properties of an operation using the token will not change from one instance of the token to another. The entities grouped under a single non-terminal may be unrelated, and there is no guarantee that the properties of an operation that uses the non-terminal will remain unchanged irrespective of which of the of the entities a particular instance refers to. It is therefore necessary, not only to identify the entity corresponding to a particular instance, but to associate with each entity a set of modifiers that may modify the properties of an operation. There are five operation properties that are of interest:

```
Token   "X:"              XMEM        { ; };
```

- Indexed Strings: These are strings that have an integer embedded in them. They refer to multiple syntactic entities which differ only in the embedded integer. The following is an example of an indexed string (taken from the description of the SPAM sample VLIW):

  ```
  Token   "ALU.R"[0..31]    ALU_R   { [0..31]; };
  ```

  Note the range operator `[0..31]` which defines the embedded integer and its range. As stated earlier, this token represents the 32 names in the group `ALU.R0`, `ALU.R1` ... `ALU.R31`.

Since it is possible for a token to represent a number of different syntactic entities, it is necessary to provide a return value identifying which syntactic entity is associated with each instance of a token. In the current version of ISDL, this return value is null or number in the case of a constant string, or a range operator in the case of an indexed string. For example, in the operation "`ALU_ADD ALU.R4, ALU.R5, ALU.R4`", the first instance of the token `ALU_R` will return the value 4, the second will return the value 5 and the third will return the value 4 again, thus successfully identifying each instance of the token. Return values are terminated by a semi-colon and are placed within "{}".

4.1.2 Other Token Related Issues

The above was an oversimplified explanation of how tokens work. It demonstrates the purpose of tokens but it omits many details concerning the use of tokens and the way operations are defined. Each operation is considered to consist of an operation name and a comma separated list of parameters. Each parameter must be either a token or a non-terminal (see Section 4.2).[6] It cannot be a constant string, for example. While this might occasionally appear awkward, it makes ISDL descriptions more uniform and easier for the tools to process. In addition, typically there is no need for constant strings in the parameter list. In the case where such a need arises, the string can be defined as a token, and the token used in the parameter list as shown in the description of the Motorola 56000, the relevant portions of which are listed below:

```
Token   X:              XMEM        { ; };
...
    DBM_MoveP XMEM, EA, XA, ACC, Y_R
```

The first line turns the string "X:" into the token `XMEM` which is then used as a parameter in the definition of the `DBM_MoveP` operation.

Note that it is an error to define two tokens with different names representing exactly the same syntactic elements. The assembler would have no way of choosing between them so it would be impossible to make use of both of them at the same time. In cases where it is necessary to have two abstractions representing the same syntactic element, one should be defined as a token and the other should be defined as a non-terminal with a single option consisting of the token (see Section 4.2.

There are also a set of predefined tokens in ISDL:

- The special token `INT`. This token will be used for any integer numerical constant that is not part of another token. Its return value is the integer constant. It will be used on integer constants in both decimal and hexadecimal notation.

- The special token `FLOAT`. This token will be used for any floating point numerical constant that is not part of another token. Its return value is the IEEE floating point representation of the constant.

[6] ISDL actually also allows the use of constant characters as a parameter. Note that this cannot refer to alphanumeric characters though, since these would be interpreted as names or integers.

4 Global Definitions

The second section of an ISDL description provides certain definitions that are used in subsequent sections of the description, most notably the Instruction Set section. The two main types of definitions provided here are Tokens and Non-Terminals. Tokens are used to group together syntactically related lexical entities in the instruction set (such as the names of registers belonging to the same register file). Non-Terminals allow the grouping of syntactically unrelated lexical entities (such as the names of registers belonging to different register files). The purpose of both tokens and non-terminals is to reduce the length of ISDL descriptions by factoring out patterns that appear multiple times. They are the main source of abstraction in ISDL.

4.1 Tokens

ISDL describes the instruction set of an architecture by describing each operation in the instruction set, and its parameters. However, in a typical instruction set, each operation might have a number of options for each parameter, and in many cases these options are syntactically related. For example, in the sample SPAM VLIW, the ALU `add` operation takes 3 arguments: the two registers whose values are to be added together and the register where the result is to be written. The value of each parameter can be any one of the registers in the ALU register file and all such registers have syntactically similar names (i.e. `ALU.R0-ALU.R31`). An example of this operation would be "`ALU_ADD ALU.R4, ALU.R5, ALU.R4`". In this case, enumerating all combinations would result in 32768 entries - one for each combination of the possible values for the parameters:

```
ALU_ADD ALU.R0, ALU.R0, ALU.R0; ...
ALU_ADD ALU.R0, ALU.R0, ALU.R1; ...
ALU_ADD ALU.R0, ALU.R1, ALU.R0; ...
...
```

Instead, it is much simpler to factor out the description of the registers in a Token, and have a single entry describing the operation by using the token in place of the parameters:

```
Token   ALU.R[0..31]    ALU_R   { [0..31]; };
```

The above line defines a token called `ALU_R` which represents the names of the registers in the register file (i.e. represents `ALU.R0-ALU.R31`). The operation definition then consists of the single entry:

```
ALU_ADD ALU_R, ALU_R, ALU_R
```

This has the same meaning as the enumeration above.

4.1.1 Syntax Of Token Definitions

Token definitions consist of four items:

1. The keyword `Token`.
2. A syntactic description of the token.
3. The token name (the name by which it will be referred to in operation definitions and non-terminal definitions)
4. A return value.

The current version of ISDL allows only two forms of syntactic descriptions:

- Constant Strings: This is just a string with no white space embedded in it. These refer to a single syntactic entity (i.e. the group of syntactic entities represented by the token has only one member). The following is an example of a constant string token definition (taken from the Motorola 56000 description).

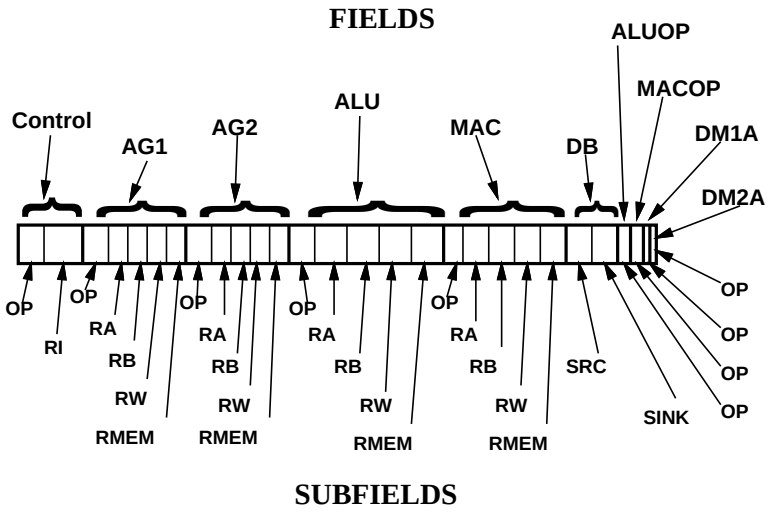

Figure 6: **The instruction word of the Sample SPAM VLIW.**

3 Format

The first of the six sections of an ISDL description, (identified by the keywords `Section Format`), is the format section. The format section defines the binary representation of the instruction word. It is divided into a number of fields (usually corresponding to the number of operations that can be performed in parallel in VLIW architectures). These are further subdivided into a number of subfields. Note that these divisions are completely arbitrary. Any division (including no division at all) could define the same word as long as the total length of the word and the order of the bits are the same. It is however beneficial to carefully divide the instruction word into fields and subfields for the following reasons:

- An appropriate division makes it easier to read, write, and modify the descriptions of how the assembly representation of an instruction corresponds to the binary image of the same instruction. For example, assume that bits 0 to 7 in the instruction word correspond to the op-code of the word and we only want to set those bits. If we have a division that makes bits 0 to 7 a subfield we can use a simple assignment to that bitfield. If on the other hand, bits 0 to 7 were part of a larger subfield or made up two smaller subfields, we would need masking operations or multiple assignments respectively.
- A proper division provides hints to the tools as to how the instruction word is conceptually organized thus making the tools simpler and faster. In particular, the disassembler and simulator are very sensitive to the division of the instruction word. Performance and code size of the tool may vary according to the division used.[5]

Example: The following is an example taken from the Sample SPAM VLIW description:

```
Section Format
Control = OP[4], RI[6];
AG2     = OP[4], RW[3], RA[3], RB[3], RMEM[3];
AG1     = OP[4], RW[3], RA[3], RB[3], RMEM[3];
ALU     = OP[4], RW[5], RA[5], RB[5], RMEM[5];
MAC     = OP[3], RW[4], RA[4], RB[4], RMEM[4];
DB      = SRC[4], SINK[4];
ALUOP   = OP[2];
MACOP   = OP[2];
DM1A    = OP[1];
DM2A    = OP[1];
```

The format section consists of one or more field declarations. Each field declaration is in the form of an assignment. The left side of the assignment is the field name. The right side of the assignment is a list of subfield declarations that make up that field. Each subfield declaration consists of a subfield name and the width of the subfield (as an integer in square brackets). Order is important both for the subfield declarations and the field declarations. It defines the order in which the fields and subfields are placed in the instruction word. The first subfield makes up the MSB of the instruction word.

The above example defines a word which is 99 bits long and has the structure shown in Figure 6.

Fields can be referred to by using their name. Subfields can be referred to using the following notation:

```
<fieldname>.<subfieldname>
```

For example, the subfield `OP` in field `Control` could be referred to as `Control.OP`. These names are only valid in the bitfield assignment portion of the Operations Section (see Section 6.1.2) and return value portion of non-terminals (see Section 4.2.1). They have no meaning elsewhere in the description.

[5] The performance of the simulator is not very sensitive to the division used since the simulator disassembles the whole binary off-line. However it is still affected by improper divisions.

2.4.4 Names

All sequences beginning with a letter and followed by one or more letters, digits, or underscores, are considered to be names. The only exceptions are keywords which may not be used as names. Names are generally used to define variables and to name various objects in the description (e.g. storage resources, fields and subfields of the instruction word, etc.) Note that a reference such as `Main.OP` (referring to a subfield in one of the example architectures) actually consists of two names, `Main` and `OP`!

2.4.5 Characters

All other characters that are not part of the above (with the exception of matching variables for constraints - see Section 7) are treated as simple characters.

2.4.1 Keywords

Most ISDL keywords are specific to the sections they appear in. Some of them however are global. The following is a list of the global keywords and their meanings:

- **Section** This in combination with one of the six section identifiers below delimits the six sections of an ISDL description. A section starts at the appearance of the keyword `Section` and the correct identifier, and ends at the beginning of the next section (the appearance of the keyword `Section` with the next identifier).
- **Format**: This is the section identifier keyword for the Format Section.
- **Global_Definitions**: This is the section identifier keyword for the Definitions Section.
- **Storage**: This is the section identifier keyword for the Storage Section.
- **Instruction_Set**: This is the section identifier keyword for the Instruction Set Section.
- **Constraints**: This is the section identifier keyword for the Constraints Section.
- **Optional**: This is the section identifier keyword for the Optional Architectural Details Section.

Note that all keywords, both global and section-specific are reserved and may not appear out of context in an ISDL description.[4]

2.4.2 Numerical Types

ISDL defines three numerical data types:

- Integers in decimal notation. These are defined as one or more digits preceded by an optional "-" sign. The number of bits an integer can represent in ISDL depend on the platform which runs the ISDL processing tools.
- Integers in hexadecimal notation. These are defined as the prefix "0x" followed by one or more digits or the characters `a-f` and `A-F`. ISDL does not allow the representation of negative numbers using hexadecimal notation since this depends on the bit-width of the storage location that will receive the value. Hexadecimals are only intended to be used as bitfield constants.
- Floating point values. These are defined as an optional "-" sign, followed by one or more digits, followed by a decimal point, followed by one or more digits, followed by an optional exponent. The syntax of the exponent is one of "e" or "E", followed by an optional "-" sign, followed by one or more digits. The range of the floating numbers that can be represented depends on the platform on which the ISDL processing tools are run.

2.4.3 Strings

Anything enclosed between "" or between "‘" is considered by ISDL to be a string and is passed to the processing tools as a single unit. Strings are not used very often in ISDL. However, Tokens and Non-Terminals (see Section 4) may have literal strings associated with them. The string type of ISDL allows these to contain ISDL keywords and other syntactic elements that ISDL processing tools would consider to be out of context thus generating an error.

The following are examples of strings:

```
"if (Section | current) {return Section;} else {return 0;}"
‘printf("Values at 0x%x: %d, %d, %d\n", addr, x, y, z);‘
```

Note that the second string includes a string in itself - ISDL forms the biggest string of either type it can.

[4] They can, however, appear in comments since comments are ignored by the ISDL parser.

Equivalently, picture the scenario where one wishes to change the width of all storage resources without having to type it in multiple times. One could declare the width using "`#define WIDTH 24`" and use the symbolic value `WIDTH` after that.

- Macro Definitions with parameters: These allow the construction of more complex textual substitutions. The have the following syntax:

  ```
  #define <macro_name>(par1, par2,...) <substitution_value>}
  ```

  The substitution text first has all instances of *par1, par2...* replaced with the actual parameter values and then the whole macro instantiation is replaced by the end result. For example:

  ```
  #define GET_MAX(x, a, b) {if (a > b) x = a; else x = b;}
  // result <- MAX(i1, 12)
  GET_MAX(result, i1, i2);
  ```

  translates to:

  ```
  {if (i1 > i2) result = i1; else result = i2;}
  ```

  This can be used to reduce complicated expressions into short statements resembling function calls.

Note that macros are only replaced in the text *following* the definition.

2.3.3 Conditional Definitions

Conditional definitions allow various portions of an ISDL description to be quickly changed or substituted by defining the appropriate value for a variable. For example, if one wants to evaluate how an architecture will behave if the multiply instruction is removed from the instruction set, one can wrap the definition of the multiply instruction in a conditional preprocessor statement:

```
// uncomment the following line to include the MPY instruction
/* #define MULTIPLY */
#ifdef MULTIPLY
    MPY a,b {...
    }
#endif
```

If the second line in the above example is commented out, the preprocessor removes the four lines following it and therefore the `MPY` instruction is effectively removed from the ISDL description. If the line is uncommented, then the preprocessor includes in the output the fourth and fifth lines, and the `MPY` instruction is included in the instruction set. In the same fashion, one can change pieces of the ISDL description at will by using the appropriate macro definitions.

2.4 Basic Types

This section deals with the basic lexical types available in ISDL and the conventions followed. It does not deal with types and keywords that are specific to each ISDL section - those are listed along with the syntax of each ISDL section.

The following types are globally defined in ISDL:

- Keywords
- Numerical types
- Names
- Strings
- Characters

In this example, the first two lines make up a multi-line comment and are ignored. The third line is actually a constraint and is processed normally. Note that multi-line comments do not *have* to occupy more than one line.

Comments cannot be nested. Comments are only meant to be a documenting aid to the people reading, writing and modifying ISDL descriptions. For example, the syntax for the descriptions of constraints (see Section 7), while rather concise and efficient, is usually slightly esoteric - it is very useful to include with each constraint a comment providing a human-readable description of the constraint.

2.3 Preprocessor Operations

All tools that use ISDL preprocess the raw ISDL file with the C preprocessor to arrive at the final ISDL description. This provides all the standard facilities of the preprocessor for use in ISDL descriptions:

- Include Mechanism
- Macro Definitions
- Conditional Definitions

Note that this also imposes some somewhat arbitrary restrictions on ISDL syntax (e.g. the character "#" is considered by the preprocessor to be a special character, and cannot be used in ISDL except to prefix preprocessor directives). A brief description of how to use the preprocessor is given below. For a more complete description read the documentation on the `cpp` preprocessor for your system.

2.3.1 Include Mechanism

While the include mechanism allows one to write an ISDL description in multiple files and then combine them into a single file, this is NOT recommended. ISDL descriptions behave as monolithic entities - i.e. changes in one section affect most of the remaining sections. Because of this it is better to organize an ISDL description as a SINGLE file. Furthermore, some tools might report erroneous line numbers when reporting errors if the include mechanism is used.

2.3.2 Macro Definitions

Macro definitions operate on the principle of textual replacement. There are two versions of definitions:

- Macro definitions with no parameters: These are generally used to replace various pieces of text with meaningful names or to make it easy to change constants that appear in multiple places. They are also useful when using conditional definitions (see Section 2.3.3). They have the following syntax:

  ```
  #define <macro_name> <substitution_value>
  ```

  For example, imagine the scenario where a set of instructions (let us say `MPY`, `ADD`, `SUB`, and `DIV`) repeatedly appear in a set of constraints in the constraints section. Rather than typing the representation of the whole group each time, one could define a macro MATH_INST, and use the macro in the constraints:

  ```
  #define MATH_INST ((MPY *) | (ADD *) | (SUB *) | (DIV *))
  // constraint such that math instructions cannot be used with jumps
  ~((JMP *) & MATH_INST)
  ~((JSR *) & MATH_INST)
  ~((JMI *) & MATH_INST)
  ```

2 ISDL Syntax

This section attempts to give a global overview of the syntax of ISDL. It does not describe in detail the syntax of each of the six sections of an ISDL description. Those details are given in the following sections. There is also a BNF description of ISDL in Appendix B.

2.1 Organization Of An ISDL Description

An ISDL description is a human-readable text file divided into six sections:

- Instruction Word Format
- Global Definitions
- Storage Resources
- Operations
- Constraints
- Optional Architectural Information

Each section begins with the keyword `Section` and one of the following keywords: `Format`, `Global_Definitions`, `Storage`, `Instruction_Set`, `Constraints`, `Optional`. Each section has its own syntax rules and keywords (described in later sections). There are, however, a number of common syntax rules and conventions that are used in all sections:

- Comments
- Preprocessor Operations
- Basic Types

2.2 Comments

ISDL allows the use of two types of comments:

- Single line comments: These begin with the string "`//`" and end at the end of the line. All characters between the above string and the next newline are considered to be a comment and are ignored. Note that this is the same as the C++ convention for single line comments. The following is an example of a single line comment:

  ```
  Control = OP[4], RI[6]; // This defines the control field
  ```

  In the above example, the statement "`Control = OP[4], RI[6];`" is a field declaration and is processed normally. The statement "`// This defines the control field`" is a single line comment and is completely ignored.

- Multi-line comments: These are enclosed between the strings "`/*`" and "`*/`". Anything between those two strings will be considered to be a comment and will be ignored. Note that this is the same as the C language convention for multi-line comments. The following is an example of a multi-line comment:

  ```
  /* Cannot drive both memories on the bus at the same time.
     Bus conflict                                              */
  ~((DM1.bus_load* *) & (DM2.bus_load* *))
  ```

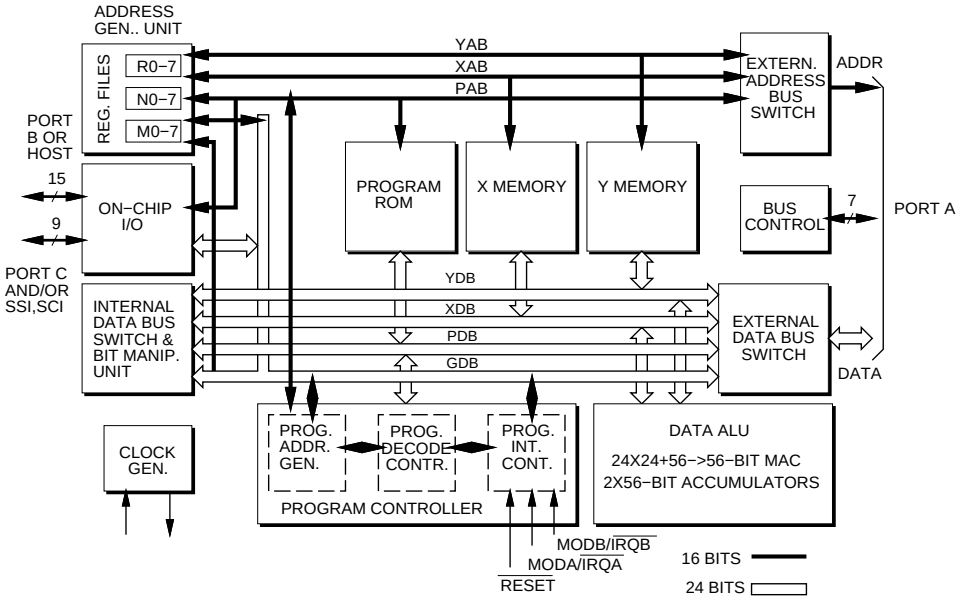

Figure 5: **The Motorola 56000 DSP engine.**

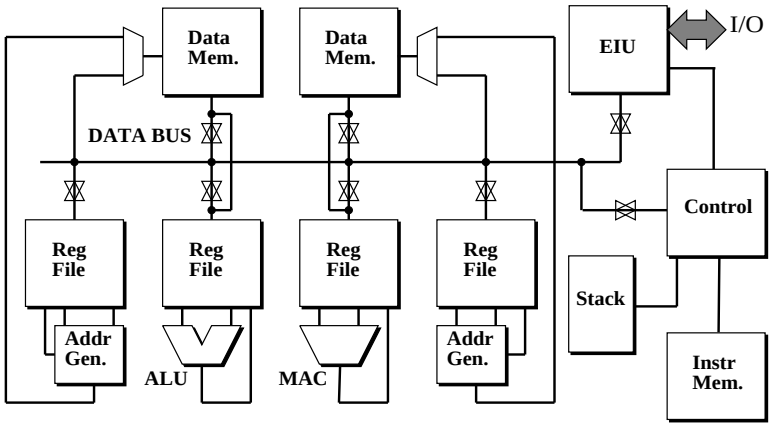

Figure 4: **The Sample SPAM VLIW processor.**

1.2 This Document

The remainder of this document is organized as follows: Section 2 describes in more detail the overall syntax of ISDL. Sections 3 through 8 describe each section of an ISDL description in more detail. In order to illustrate various points they use as examples parts of the descriptions of two sample architectures:

- The Sample SPAM VLIW. This architecture (shown in Figure 4) is an example VLIW architecture that was designed internally to allow us to explore the limits of ISDL and our various tools. It is therefore a rather aggressive architecture which is not necessarily well suited for any particular application. It combines features from DSPs (like the use of data transfers in parallel with actual computation and the use of address generators) with features from RISC processors (such as a load-store architecture).
- The Motorola 56000 DSP engine (shown in Figure 5). This is an example of a commercial DSP currently in use. Although not as ambitious as the Sample SPAM VLIW, it is much more complex in many aspects (especially since it uses heavy encoding to shrink the instruction word size to 24 bits).

The complete descriptions of both architectures can be found in Appendix C. Note that even though the SPAM VLIW has many more functional units and instructions than the 56000 DSP, the description of the 56000 DSP is much longer than that of the SPAM VLIW simply because of the complexity of the instruction set of the former.

Section 9 describes additional hints as to how to write good ISDL descriptions. By following the advice in this section, one can create more intuitive descriptions and/or write descriptions that are easier for the tools to work with. Section 10 describes improvements to ISDL that are forthcoming, as well as other enhancements that we might not necessarily incorporate any time soon. Appendix A contains a glossary of terms that might be useful when reading this document.

1.1.2 Automatic Generation Of Support Tools

ISDL is specifically designed to support the generation of four main design support tools:

- Code Generator
- Assembler
- Disassembler
- Instruction Level Simulator (ILS)

The code generator and assembler, along with a compiler front end, make up a retargetable compiler system whose target architecture can be changed through ISDL. The disassembler is mainly provided as a convenience tool. The Instruction Level Simulator is used to measure the performance of the system, verify manual changes to the code, and perform preliminary testing of the system.

ISDL contains enough information to generate all of the above tools automatically. Furthermore, it contains enough information to allow the code generator to perform optimizations customized to the particular target architecture.

1.1.3 ISDL Descriptions As A Programmer's Manual

In order to make the task of understanding, writing and modifying ISDL descriptions easier, ISDL was designed to look like the conventional Programmer's Manual that accompanies most commercial processors and DSPs. It turns out that this also makes it easier to generate the support tools. In particular, ISDL models an architecture by describing the user visible state and then listing every operation that can affect this state. Thus, ISDL is a behavioral language (rather than a structural one). It does, however, contain enough structural information to allow for the complete description of an architecture (although it tends to disguise such structural information in a behavioral-like fashion).

An ISDL description consists of 6 sections:

- Instruction Word Format
- Global Definitions
- Storage Resources
- Operations
- Constraints
- Optional Architectural Information

The Instruction Word Format describes how long the instruction word is, and how it can be broken down into subfields. Global Definitions correspond to the various data types and tables of common sub-expressions (such as addressing modes) common in most processors and DSPs. They are the main source of abstraction in ISDL. The Storage Resources section explicitly lists all the visible state of the system.[3] The Operations section then lists all possible operations, (grouped in *fields*), that the architecture provides. The effect of the operation on visible state, the assembly representation of the operation, the binary representation of the operation, the timing characteristics of each operation, and various costs are all listed on a per-operation basis. The Constraints section describes various restrictions as to how these operations may be assembled into instruction words and programs. This corresponds to the restrictions commonly listed in the Programmer's Manual for each instruction.

[3] Visible state includes control registers and memory-mapped I/O.

This methodology itself (called architecture exploration by iterative improvement), can be embedded in the Hardware/Software Co-Design[2][3] paradigm to arrive at even more effective solutions. In this paradigm, the first step is to create a description of the application in a high-level programming language (e.g. C or C++). This can be directly compiled and run on a workstation to verify that the description is indeed a correct representation of the required behavior. Once the designer is sure that the description corresponds to the required task, the most time-critical tasks of the resulting code are removed and synthesized into custom hardware. The remainder is left as software at this step, which is known as the partitioning step. The remaining software needs to be run on a processor of some sort. Such a processor is created, either by hand or automatically using another tool, and the resulting architecture described in ISDL. This architecture is then evaluated and modified as appropriate by using the design methodology of Figure 2. If the modification iterations do not result in a satisfactory architecture (one that meets the design criteria of cost, power consumption, performance, etc.) then the design is repartitioned and the whole process repeated again. The Hardware/Software Co-Design methodology is shown in Figure 3.

1.1 Requirements

The target application of ISDL implies the following attributes:

- ISDL must be able to specify a wide variety of architectures.
- ISDL must support the automatic generation of at least an assembler, a disassembler, an instruction level simulator and a retargetable compiler, and possibly other tools as well.
- ISDL must provide enough information for the compiler back end to be able to optimize the code for the particular architecture at hand.
- ISDL descriptions must be easy for an engineer to write, understand and modify.

1.1.1 Specifying A Wide Variety Of Architectures

It is important that the range of architectures that are supported is not limited by the abilities of ISDL but rather by the abilities of the various tools that handle ISDL descriptions and those that are generated from ISDL descriptions. In order to achieve this, ISDL attempts to encompass as wide a variety of architectures as possible. In particular, ISDL supports Very Long Instruction Word (VLIW) architectures; these are not explicitly supported by most other machine description languages, and are a superset of more traditional architectures. VLIW architectures have more than one functional unit, and these functional units can be used in parallel. This parallelism is reflected in the instruction set, where instructions are groups of operations that can be performed in parallel.[2] One can consider the more traditional architectures that only have one unit active at any given time in their instruction set, (from now on called *unifunctional* architectures), to be degenerate cases of VLIW architectures.

VLIW architectures are very important for two reasons:

- By amortizing control overhead over a number of functional units they actually result in more efficient use of silicon area.
- Given that the set of VLIW architectures can be considered as a superset of the set of unifunctional architectures, then if we can describe VLIW architectures, we can automatically describe unifunctional architectures.

ISDL has specific features to support the efficient description of VLIW architectures. It can also gracefully handle unifunctional architectures. These two classes of architectures cover traditional micro-controllers, CISC and RISC processors, vector processors, most DSP cores, and a large number of Application Specific Instruction-set Processors (ASIPs).

[2] This is in contrast to parallelism which is not apparent in the instruction set (such as the parallelism in super-scalar architectures).

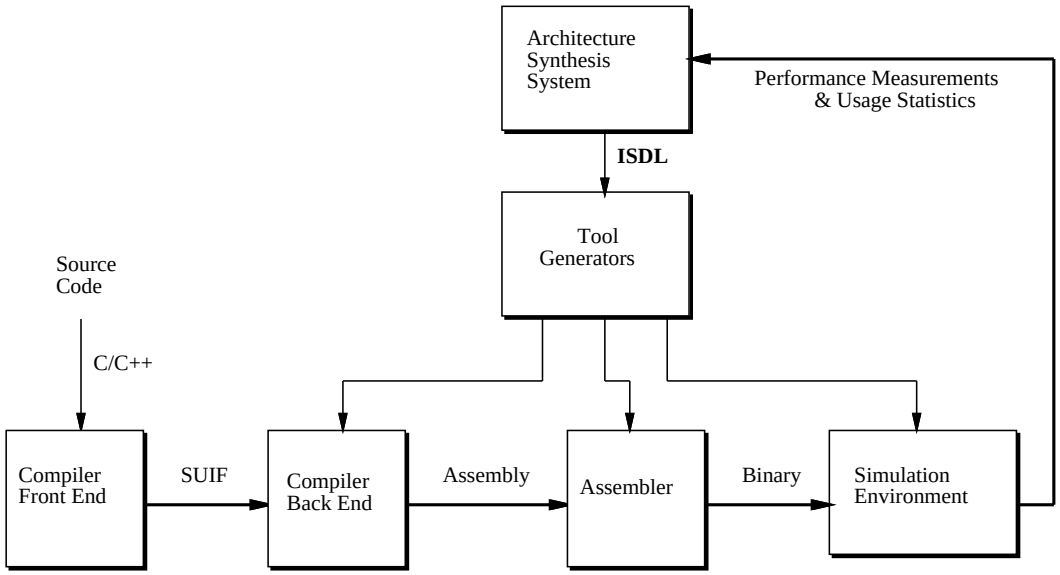

Figure 2: **Architecture exploration using ISDL.**

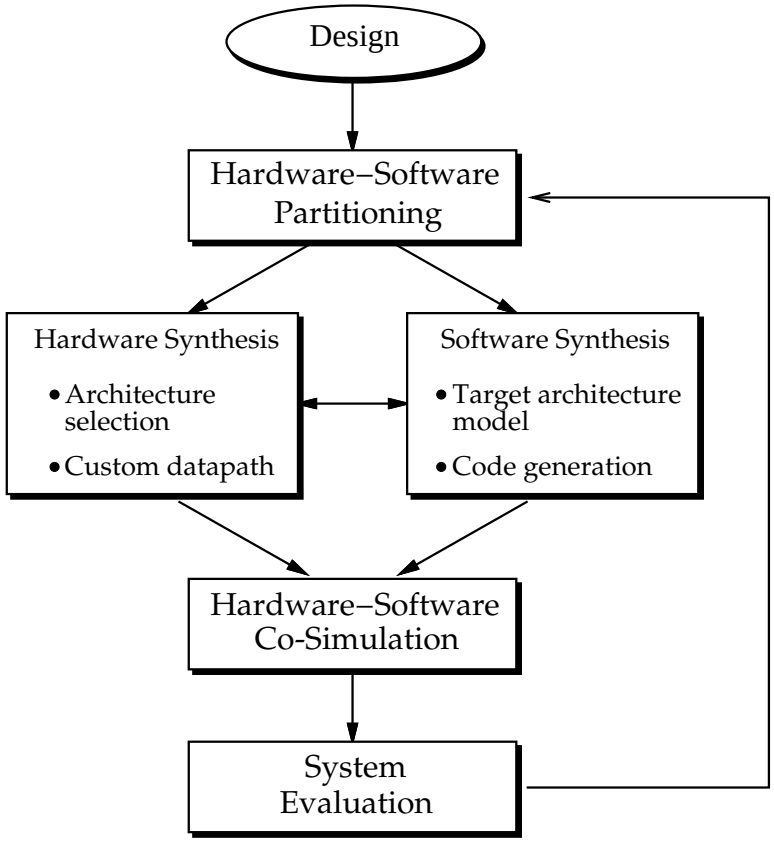

Figure 3: **The Hardware/Software Co-design Methodology.**

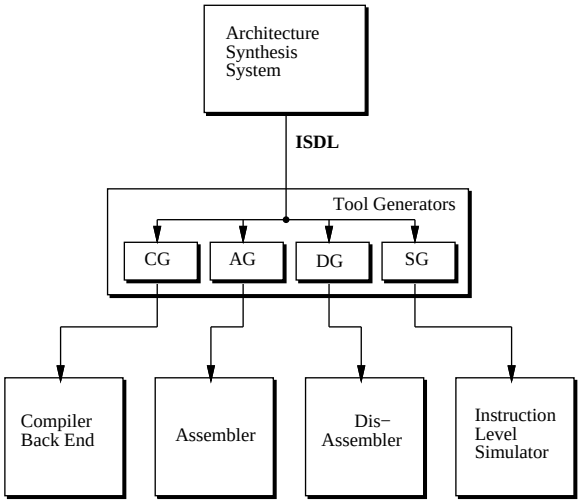

Figure 1: **ISDL and generated tools.**

1 Introduction

ISDL (Instruction Set Description Language) is a machine description language specifically designed to support retargetable tools for architecture exploration and development. In particular, it is designed to allow the automatic generation of an assembler, disassembler, code generator and instruction level simulator from a single description of the architecture. At the same time, it is designed to support architecture descriptions that are either generated automatically by another tool or generated and/or modified by hand by an engineer. Figure 1 shows how ISDL descriptions can be used to generate a set of tools to support a design environment.

ISDL is most useful in applications where cost and manufacturing concerns dictate that a custom architecture should be used for a particular application[1] but, at the same time, short design cycles are required. Most embedded system designs fall in this category. Usually, cost and size are a primary concern for such systems, so custom architectures are preferred. Unfortunately, time to market and therefore short design cycles are equally important. This invariably means that there is no time to develop tools to support the design environment for each architecture. This limits architecture exploration and may force the use of existing off-the-shelf solutions even if they are not well suited to the application at hand. ISDL, by allowing for the automatic generation of the support tools, eliminates the pressure to use off-the-shelf solutions and allows effective exploration of the architecture design space[1].

Figure 2 illustrates a design methodology which is well suited for such applications. In this design methodology, an engineer or another tool generates an ISDL description of an architecture customized for a particular application. This description is then used to generate an assembler, a disassembler, an instruction level simulator and a code generator. The code generator can be used as the back end of a compiler, thus resulting in a retargetable compiler. These tools can then be used to evaluate the target architecture in the context of the target application. To do so, the application source code is compiled using the retargetable compiler. The output of the retargetable compiler is an assembly program which can be further tuned by a human engineer if necessary. The assembly is then converted to an executable binary using the automatically generated assembler. This binary is then passed to the simulator which executes the program and allows various performance measurements to be made. If the performance of the system is not adequate or system cost can be reduced without sacrificing performance, the architecture is modified on the basis of the measurements and a new ISDL description is generated for the new architecture. The whole process is repeated until a suitable architecture is found.

[1] as opposed to an off-the-shelf solution, such as a DSP core or commercial microprocessor.

List of Figures

Contents

Acknowledgments

This document is the result of the efforts of a number of people other than myself. I would first of all like to thank Silvina Z. Hanono for the time she spend working with me on the structure and initial version of ISDL, and for her help in producing this document. I would also like to thank Prof. Srinivas Devadas for giving this project its direction and for constant advice concerning likely pitfalls. Finally I would like to thank Daniel Engels for sanity checking my ideas and for his help in producing this document.

Copyrights and Trademarks

ISDL: Instruction Set Description Language
Version 1.0

George Hadjiyiannis
SPAM
MIT RLE

November 5, 1998