1 Hardware

The computer is composed of a largely isolated data unit and control unit, which are only connected by a couple of direct buses.

1.1 Data path

All components of the data path have both 16-bit word size and address length. They are connected as such:
The system’s storage components adhere to these specifications:

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>counter register</td>
<td>program counter</td>
</tr>
<tr>
<td>IR</td>
<td>shift register†</td>
<td>instruction register</td>
</tr>
<tr>
<td>MDR</td>
<td>shift register</td>
<td>main data register</td>
</tr>
<tr>
<td>TMP</td>
<td>register</td>
<td>temporary register</td>
</tr>
<tr>
<td>Mem</td>
<td>memory</td>
<td>main memory (program and data)</td>
</tr>
<tr>
<td>Reg</td>
<td>16-register bank</td>
<td>general-purpose registers (see ISA document, section 5)</td>
</tr>
<tr>
<td>Val</td>
<td>3-register bank</td>
<td>storage of immediate values</td>
</tr>
<tr>
<td>Addr</td>
<td>3-register bank</td>
<td>storage of effective addresses</td>
</tr>
</tbody>
</table>

### 1.2 Control path

The word size and address widths within the control path are component-specific:

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>word size</th>
<th>purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>µPC</td>
<td>counter register</td>
<td>12-bit</td>
<td>microprogram counter</td>
</tr>
<tr>
<td>µIR</td>
<td>register</td>
<td>16-bit</td>
<td>microinstruction register</td>
</tr>
<tr>
<td>cntrl</td>
<td>3-register bank†</td>
<td>5-bit</td>
<td>operand control flags</td>
</tr>
<tr>
<td>tmp</td>
<td>register</td>
<td>5-bit</td>
<td>temporary control flags</td>
</tr>
<tr>
<td>i</td>
<td>counter register</td>
<td>2-bit</td>
<td>track current working operand</td>
</tr>
<tr>
<td>regshift</td>
<td>shift register</td>
<td>16-bit</td>
<td>transfer register IDs from data path</td>
</tr>
<tr>
<td>µSP</td>
<td>counter register</td>
<td>2-bit</td>
<td>micro-stack pointer</td>
</tr>
<tr>
<td>µStack</td>
<td>3-register bank</td>
<td>12-bit</td>
<td>subprocedure return addresses</td>
</tr>
<tr>
<td>µMem</td>
<td>memory</td>
<td>16-bit data, 12-bit addr</td>
<td>contains hardcoded microprogram</td>
</tr>
<tr>
<td>µJumpTab</td>
<td>memory</td>
<td>12-bit data, 7-bit addr</td>
<td>hardcoded jump table</td>
</tr>
</tbody>
</table>

**Control path buses**

<table>
<thead>
<tr>
<th>name</th>
<th>length</th>
<th>purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>µABUS</td>
<td>12</td>
<td>Moves the address of microcode</td>
</tr>
<tr>
<td>µJBUS</td>
<td>7</td>
<td>Moves around the information for jumping</td>
</tr>
<tr>
<td>pBUS</td>
<td>4</td>
<td>Moves the register number into the cntrl registers</td>
</tr>
<tr>
<td>pDBUS</td>
<td>5</td>
<td>Moves the cntrl values between cntrl registers and tmp</td>
</tr>
</tbody>
</table>

The 3 control registers are used to represent the source and destination operand locations:

<table>
<thead>
<tr>
<th>msb</th>
<th>trailing bits</th>
<th>significance</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4-bit register identifier</td>
<td>this operand/destination is located in the specified GPR</td>
</tr>
<tr>
<td>1</td>
<td>0000</td>
<td>this operand is not used by the current operation</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>this operand is located in register Val[1]</td>
</tr>
<tr>
<td>1</td>
<td>0010</td>
<td>this operand is located in register Val[2]</td>
</tr>
<tr>
<td>1</td>
<td>0011</td>
<td>this operand is located in the MDR</td>
</tr>
<tr>
<td>1</td>
<td>0100</td>
<td>this operand/destination is located at memory address Addr[0]</td>
</tr>
<tr>
<td>1</td>
<td>0101</td>
<td>this operand is located at memory address Addr[1]</td>
</tr>
<tr>
<td>1</td>
<td>0110</td>
<td>this operand is located at memory address Addr[2]</td>
</tr>
</tbody>
</table>

†These flippable shift registers support hardware-based toggling of individual bits.
2 Register transfer language

Here is the sequence of hardware actions performed by each phase of instruction processing:

2.1 Fetch phase

IR <- Mem[PC] # the instruction word
PC <- PC + 1

2.2 Decode phase

i <- 00
loop
    if i != 00 AND IR(imm)(i) = 1 then
        Val[i] <- Mem[PC]
        cntl[i] <- 10000 | i
        PC <- PC + 1
    elif IR(ami) = 00 then
cntl[i] <- 10000
elif IR(ami) = 01 or 10 then
    Addr[i] <- Mem[PC]
    cntl[i] <- 10100 | i
    PC <- PC + 1
elif IR(ami) = 11 then
    MDR <- Mem[PC] # the R-type immediate word
    cntl[i] <- 10100 | i
    PC <- PC + 1
if MDR(sam) = 000 then # register value
    regshift <- MDR
    regshift <- regshift >> 8 # reg0
    cntl[i] <- 00000 | regshift
elif MDR(sam) = 001 then # register indirect
    Addr[i] <- Reg[MDR(reg0)]
elif MDR(sam) = 010 then # scaled
    Val[i] <- MDR
    MDR <- Mem[PC] # the S-type immediate
    PC <- PC + 1
    MDR <- MDR >> 8
    Addr[i] <- Reg[Val[i](reg1)] << MDR
    Addr[i] <- Reg[Val[i](reg0)] + Addr[i]
elif MDR(sam) = 011 then # doubly scaled
    Val[i] <- MDR
    MDR <- Mem[PC]
    TMP <- MDR
    PC <- PC + 1
    MDR <- MDR >> 8
    Addr[i] <- Reg[Val[i](reg1)] << MDR
    Addr[i] <- Reg[Val[i](reg0)] + Addr[i]
    MDR <- Mem[Addr[i]]
    Addr[i] <- MDR
    MDR <- TMP & 0xff
    MDR <- Reg[Val[i](reg2)] << MDR
    Addr[i] <- Addr[i] + MDR
elif MDR(sam) = 100 then # auto increment
    Val[i] <- MDR
    Addr[i] <- Reg[Val[i](reg0)]
    Reg[Val[i](reg0)] <- Reg[Val[i](reg0)] + 1
elif MDR(sam) = 101 then # auto decrement
    Val[i] <- MDR
    Addr[i] <- Reg[Val[i](reg0)]
    Reg[Val[i](reg0)] <- Reg[Val[i](reg0)] - 1
elif MDR(sam) = 110 then # scaled displacement
    Val[i] <- MDR
    MDR <- Mem[PC] # the S-type immediate
    PC <- PC + 1
MDR <- MDR >> 8
Addr[i] <- Reg[Val[i](reg0)] << MDR
MDR <- Mem[PC]  # the I-type immediate
PC <- PC + 1
Addr[i] <- Addr[i] + MDR

elif MDR(sam) = 111 then  # doubly scaled displacement
  Val[i] <- MDR
  MDR <- Mem[PC]  # the S-type immediate
  PC <- PC + 1
  TMP <- MDR
  MDR <- MDR >> 8
  Addr[i] <- Reg[Val[i](reg0)] << MDR
  MDR <- Mem[PC]  # the I-type immediate
  PC <- PC + 1
  Addr[i] <- Addr[i] + MDR
  MDR <- Mem[Addr[i]]
  Addr[i] <- MDR
  MDR <- TMP & 0xff
  MDR <- Reg[Val[i](reg1)] << MDR
  Addr[i] <- Addr[i] + MDR
fi

fi
i <- i + 1
until i = 11 repeat

i <- 00
loop
  if IR(ami) = 10 then  # PC-relative
    Addr[i] <- Addr[i] + PC
  fi
  i <- i + 1
until i = 11 repeat

2.3 Memory Load

i <- 00
loop
  if cntl[i](4) = 1 AND cntl[i](2) = 1 then
    Val[i] <- Mem[Addr[i]]
    if i != 00 then
      cntl[i](2) <- 0
  fi
fi
i <- i + 1
until i = 11 repeat
2.4 Execute

# call function at uJumpTab label IR(opc)

2.5 Writeback

if cntl[0](4) = 0 then
    Reg[cntl[0](3..0)] <- MDR
elif cntl[0](2) = 1 then
    Mem[Addr[cntl[0](3..0)]] <- MDR
fi

# jump to the very beginning

2.6 Supporting Functions

halt:
    # bail out

and:
    # call validator_one
    MDR <- op1 & op2
    # return

or:
    # call validator_one
    MDR <- op1 | op2
    # return

xor:
    # call validator_one
    MDR <- op1 ^ op2
    # return

lsft:
    # call validator_one
    MDR <- op1 << op2
    # return

nand:
    # call validator_one
    # call and
    cntl[1] <- 10011
    cntl[2] <- 10000
    # call not
    # return

nor:
    # call validator_one
    # call or
    cntl[1] <- 10011
    cntl[2] <- 10000
    # call not
    # return

xnor:
# call validator_one
# call xor
cntl[1] <- 10011
cntl[2] <- 10000

# call not
# return

rsft:
    # call validator_one
    MDR <- op1 >> op2
    # return
    # logical goes here
    # logical goes here
    # logical goes here

rasft:
    # call validator_one
    MDR <- op1 >>> op2
    # return
    # illogical goes here
    # illogical goes here
    # illogical goes here
    # illogical goes here

slt:
    # call validator_one
    # call sub
    if MDR(15) = 1 then
        MDR <- 1
    else
        MDR <- 0
    fi
    # return

sgt:
    # call validator_one
    MDR <- op2 - op1
    if MDR(15) = 1 then
        MDR <- 1
    else
        MDR <- 0
    fi
    # return

seq:
    # call validator_one
    # call sub
    if MDR = 0 then
        MDR <- 1
    else
        MDR <- 0
    fi
    # return
sne:
# call validator_one
# call seq
cntl[1] <- 10011
cntl[2] <- 10000
# call siz
# return

sle:
# call validator_one
# call sgt
cntl[1] <- 10011
cntl[2] <- 10000
# call siz
# return

sgs:
# call validator_one
# call slt
cntl[1] <- 10011
cntl[2] <- 10000
# call siz
# return

add:
# call validator_one
MDR <- op1 + op2
# return

sub:
# call validator_one
MDR <- op1 - op2
# return
# compbranch goes here
# compbranch goes here
# compbranch goes here
# compbranch goes here
# compbranch goes here
# compbranch goes here
prnt:
# call validator_three
# print out op1
# return
# siz goes here
siz: # handles lnot and siz
# call validator_four
if op1 = 0 then
  MDR <- 1
else
  MDR <- 0
fi
# return

**snz:**

# call validator_four
# call siz
cntl[1] <- 10011
# call siz
# return

**not:**

# call validator_four
MDR <- ~ op1
# return

**neg:**

# call validator_four
MDR <- 0 - op1
# return

# simpbranch goes here
# simpbranch goes here

**incr:**

# call validator_six
# call add
# return

**decr:**

# call validator_six
# call sub
# return

**jmp:**

# call validator_seven
PC <- op0
cntl[0] <- 10011
# return

**jal:**

# call validator_seven
Reg[15] <- PC
PC <- op0
cntl[0] <- 10011
# return

**call:**

# call validator_seven
# call jal
Mem[Reg[14]] <- Reg[15]
# return

**ret:**

# call validator_eight
Addr[0] <- Mem[Reg[14]]
cntl[0] <- 10100
# call jmp
# return

move:
# call validator_nine
MDR <- op1
# return

logical: # handles land, lor, lxor
  tmp <- cntl[2]
cntl[2] <- 10000
# call snz
Val[1] <- MDR
cntl[1] <- tmp
cntl[2] <- 10000
# call snz
cntl[1] <- 10001
cntl[2] <- 10011
IR(opc)(3) <- 0 # IR(opc) - 8
# call function at uJumpTab label IR(opc)
# return

illogical: # handles lnand, lnor, lxnor
  IR(opc)(2) <- 0 # IR(opc) - 4
# call logical
cntl[1] <- 10011
cntl[2] <- 10000
# call siz
# return

compbranch: # handles blt, bgt, beq, bne, ble, bge
# call validator_two
IR(opc)(3) <- 0 # IR(opc) - 8
# call function at uJumpTab label IR(opc)
if MDR = 1 then
  PC <- op0
fi
cntl[0] <- 10011
# return

simpbranch: # handles biz, bnz
# call validator_five
IR(opc)(2) <- 0 # IR(opc) - 4
# call function at uJumpTab label IR(opc)
if MDR = 1 then
  PC <- op0
fi
cntl[0] <- 10011
validator_one:
    # Check for 2-3 ops, and detect/handle shorthand form
    if cntl[0] = 10000 then
        # bail out loudly
        fi
    if cntl[1] = 10000 then
        # bail out loudly
        fi
    if cntl[2] = 10000 then
        cntl[1] <- cntl[0]
        if cntl[1](4) = 1 AND cntl[1](2) = 1 then
            cntl[1](2) <- 0
        fi
    fi
    # return

validator_two:
    # Check for 3 ops, op0 is not a register
    if cntl[0] = 10000 then
        # bail out loudly
        fi
    if cntl[1] = 10000 then
        # bail out loudly
        fi
    if cntl[2] = 10000 then
        # bail out loudly
        fi
    if cntl[0](4) = 0 then
        # bail out loudly
        fi
    # return

validator_three:
    # Check for 1 op, and shuttle it into position 1
    if cntl[0] = 10000 then
        # bail out loudly
        fi
    cntl[1] <- cntl[0]
    cntl[0] <- 10011
    if cntl[1](4) = 1 AND cntl[1](2) = 1 then
        cntl[1](2) <- 0
    fi
    # return

# return
validator_four:
    # Check for 1-2 ops, set cntl[1] if it was empty
    if cntl[0] = 10000 then
        # bail out loudly
        fi
    if cntl[1] = 10000 then
        cntl[1] <- cntl[0]
        if cntl[1](4) = 1 AND cntl[1](2) = 1 then
            cntl[1](2) <- 0
        fi
    fi
    # return

validator_five:
    # Check for 2 ops, op0 is not a register
    if cntl[0] = 10000 then
        # bail out loudly
        fi
    if cntl[1] = 10000 then
        # bail out loudly
        fi
    if cntl[0](4) = 0 then
        # bail out loudly
        fi
    # return

validator_six:
    # Check for 1 op, and prepare for binary operation with 1
    if cntl[0] = 10000 then
        # bail out loudly
        fi
    MDR <- 1
    cntl[1] <- 10011
    # return

validator_seven:
    # Check for 1 op, op0 is not a register
    if cntl[0] = 10000 then
        # bail out loudly
        fi
    if cntl[0](4) = 0 then
        # bail out loudly
        fi
    # return

validator_eight:
    # Check for 0 op
validator_nine:
    # Check for 2 ops
    if cntl[0] = 10000 then
        # bail out loudly
    fi
    if cntl[1] = 10000 then
        # bail out loudly
    fi
    # return

3 Microinstruction format

Each microinstruction is encoded as a single word, with one of two possible formats:

<table>
<thead>
<tr>
<th>C-type word format</th>
<th>J-type word format</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>control points</td>
</tr>
<tr>
<td>15 12</td>
<td>15 12</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

3.1 C-type microinstructions

This microinstruction type is used to set control points and move data around in the data and control paths. The encoding is entirely vertical and therefore supports no parallelism beyond that encoded into the discrete control point identifiers themselves.

<table>
<thead>
<tr>
<th>encoding</th>
<th>equivalent RTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>Addr[0] ← Mem[Reg[14]]</td>
</tr>
<tr>
<td>0x01</td>
<td>Addr[i] ← Addr[i] + MDR</td>
</tr>
<tr>
<td>0x02</td>
<td>Addr[i] ← Addr[i] + PC</td>
</tr>
<tr>
<td>0x03</td>
<td>Addr[i] ← MDR</td>
</tr>
<tr>
<td>0x04</td>
<td>Addr[i] ← Mem[PC]</td>
</tr>
<tr>
<td>0x05</td>
<td>Addr[i] ← Reg[MDR(reg0)]</td>
</tr>
<tr>
<td>0x06</td>
<td>Addr[i] ← Reg[Val<a href="reg0">i</a>]</td>
</tr>
<tr>
<td>0x07</td>
<td>Addr[i] ← Reg[Val<a href="reg0">i</a>] + Addr[i]</td>
</tr>
<tr>
<td>0x08</td>
<td>Addr[i] ← Reg[Val<a href="reg0">i</a>] &lt; &lt; MDR</td>
</tr>
<tr>
<td>0x09</td>
<td>Addr[i] ← Reg[Val<a href="reg1">i</a>] &lt; &lt; MDR</td>
</tr>
<tr>
<td>0xa</td>
<td>IR ← Mem[PC]</td>
</tr>
<tr>
<td>0xb</td>
<td>IR(opc)(2) ← 0</td>
</tr>
<tr>
<td>0xc</td>
<td>IR(opc)(3) ← 0</td>
</tr>
<tr>
<td>0xd</td>
<td>MDR ← 0</td>
</tr>
<tr>
<td>0xe</td>
<td>MDR ← 0 - op1</td>
</tr>
<tr>
<td>0xf</td>
<td>MDR ← 1</td>
</tr>
<tr>
<td>0x10</td>
<td>MDR ← MDR &gt;&gt; 8</td>
</tr>
<tr>
<td>0x11</td>
<td>MDR ← Mem[Addr[i]]</td>
</tr>
<tr>
<td>0x12</td>
<td>MDR ← Mem[PC]</td>
</tr>
<tr>
<td>0x13</td>
<td>MDR ← Reg[Val<a href="reg1">i</a>] &lt; &lt; MDR</td>
</tr>
</tbody>
</table>
0x14  MDR ← Reg[Val[i](reg2)] << MDR
0x15  MDR ← TMP & 0xff
0x16  MDR ← op1
0x17  MDR ← op1 & op2
0x18  MDR ← op1 + op2
0x19  MDR ← op1 - op2
0x1a  MDR ← op1 << op2
0x1b  MDR ← op1 >> op2
0x1c  MDR ← op1 >> > op2
0x1d  MDR ← op1 ^ op2
0x1e  MDR ← op1 | op2
0x1f  MDR ← op2 - op1
0x20  MDR ← ~ op1
0x21  Mem[Addr[cntl][0](3..0)] ← MDR
0x22  Mem[Reg[14]] ← Reg[15]
0x23  PC ← PC + 1
0x24  PC ← op0
0x25  Reg[14] ← Reg[14] + 1
0x26  Reg[14] ← Reg[14] - 1
0x27  Reg[15] ← PC
0x28  Reg[Val[i](reg0)] ← Reg[Val[i](reg0)] + 1
0x29  Reg[Val[i](reg0)] ← Reg[Val[i](reg0)] - 1
0x2a  Reg[cntl][0](3..0) ← MDR
0x2b  TMP ← MDR
0x2c  Val[i] ← MDR
0x2d  Val[i] ← MDR
0x2e  Val[i] ← Mem[Addr[i]]
0x2f  Val[i] ← Mem[PC]
0x30  cntl[0] ← 10011
0x31  cntl[0] ← 10100
0x32  cntl[1] ← 10001
0x33  cntl[1] ← 10011
0x34  cntl[1] ← cntl[0]
0x35  cntl[1] ← tmp
0x36  cntl[1](2) ← 0
0x37  cntl[2] ← 10000
0x38  cntl[2] ← 10011
0x3a  cntl[i] ← 00000 | regshift
0x3b  cntl[i] ← 10000
0x3c  cntl[i] ← 10000 | i
0x3d  cntl[i] ← 10100 | i
0x3e  cntl[i](2) ← 0
0x3f  i ← 00
0x40  i ← i + 1
0x41  regshift ← MDR
0x42  regshift ← regshift >> 8
3.2 J-type microinstructions

This microinstruction format is used for goto operations altering the micro-program counter and microstack. The type field determines what type of control flow change is occurring, as well as which of the immediates will actually be used.

<table>
<thead>
<tr>
<th>encoding</th>
<th>function</th>
<th>condition and jump fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>(C-type microinstruction)</td>
<td>N/A</td>
</tr>
<tr>
<td>0x1</td>
<td>jump to jump table label</td>
<td>only jump used</td>
</tr>
<tr>
<td>0x2</td>
<td>jump to beginning of the microprogram</td>
<td>both ignored</td>
</tr>
<tr>
<td>0x3</td>
<td>call function at jump table label</td>
<td>only jump used</td>
</tr>
<tr>
<td>0x4</td>
<td>call function at jump table address IR(opc)</td>
<td>both ignored</td>
</tr>
<tr>
<td>0x5</td>
<td>return from a call</td>
<td>both ignored</td>
</tr>
<tr>
<td>0x6</td>
<td>halt system normally</td>
<td>both ignored</td>
</tr>
<tr>
<td>0x7</td>
<td>halt system due to failed operand validation</td>
<td>both ignored</td>
</tr>
<tr>
<td>0x8</td>
<td>print contents of op1</td>
<td>both ignored</td>
</tr>
<tr>
<td>0x9</td>
<td>(invalid)</td>
<td>N/A</td>
</tr>
<tr>
<td>0xa</td>
<td>branch to jump table label</td>
<td>both used</td>
</tr>
<tr>
<td>0xb</td>
<td>branch to beginning of the microprogram</td>
<td>both ignored</td>
</tr>
<tr>
<td>0xc</td>
<td>conditionally call function by jump table</td>
<td>both used</td>
</tr>
<tr>
<td>0xd</td>
<td>conditionally call function at IR(opc)</td>
<td>only condition used</td>
</tr>
<tr>
<td>0xe</td>
<td>conditionally return from a call</td>
<td>only condition used</td>
</tr>
</tbody>
</table>
If the goto is conditional, the condition bits determine the sufficient clause as follows:

<table>
<thead>
<tr>
<th>encoding</th>
<th>expansion</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>IR(ami) = 00</td>
</tr>
<tr>
<td>0x01</td>
<td>IR(ami) = 01 or 10</td>
</tr>
<tr>
<td>0x02</td>
<td>IR(ami) = 10</td>
</tr>
<tr>
<td>0x03</td>
<td>IR(ami) = 11</td>
</tr>
<tr>
<td>0x04</td>
<td>MDR = 0</td>
</tr>
<tr>
<td>0x05</td>
<td>MDR = 1</td>
</tr>
<tr>
<td>0x06</td>
<td>MDR(15) = 1</td>
</tr>
<tr>
<td>0x07</td>
<td>MDR(sam) = 000</td>
</tr>
<tr>
<td>0x08</td>
<td>MDR(sam) = 001</td>
</tr>
<tr>
<td>0x09</td>
<td>MDR(sam) = 010</td>
</tr>
<tr>
<td>0x0a</td>
<td>MDR(sam) = 011</td>
</tr>
<tr>
<td>0x0b</td>
<td>MDR(sam) = 100</td>
</tr>
<tr>
<td>0x0c</td>
<td>MDR(sam) = 101</td>
</tr>
<tr>
<td>0x0d</td>
<td>MDR(sam) = 110</td>
</tr>
<tr>
<td>0x0e</td>
<td>MDR(sam) = 111</td>
</tr>
<tr>
<td>0x0f</td>
<td>cntl[0] = 10000</td>
</tr>
<tr>
<td>0x10</td>
<td>cntl<a href="2">0</a> = 1</td>
</tr>
<tr>
<td>0x11</td>
<td>cntl<a href="4">0</a> = 0</td>
</tr>
<tr>
<td>0x12</td>
<td>cntl[1] = 10000</td>
</tr>
<tr>
<td>0x13</td>
<td>cntl<a href="4">1</a> = 1 AND cntl<a href="2">1</a> = 1</td>
</tr>
<tr>
<td>0x14</td>
<td>cntl[2] = 10000</td>
</tr>
<tr>
<td>0x15</td>
<td>cntl<a href="4">i</a> = 1 AND cntl<a href="2">i</a> = 1</td>
</tr>
<tr>
<td>0x16</td>
<td>i != 00</td>
</tr>
<tr>
<td>0x17</td>
<td>i != 00 AND IR(imm)(i) = 1</td>
</tr>
<tr>
<td>0x18</td>
<td>i = 11</td>
</tr>
<tr>
<td>0x19</td>
<td>op1 = 0</td>
</tr>
</tbody>
</table>

Jump destinations are encoded as addresses in the jump table, which is stored in a dedicated memory module within the control unit. Each location therein is analogous to a label, and contains a microprogram memory address. The precise number and ordering of labels within this table are unspecified, except that the lowest addresses are to be used for the user-facing instruction opcodes in the exact order enumerated under section 3.1 of the ISA document. This requirement is imposed to allow efficient decoding of user-generated instruction words.