Assembling Branch Instructions  (chapter 3)

Branch beq $rs,$rt,offset16  
$pc = ($rt == $rs)? ($pc+4+(δ32(offset16)<<2))):($pc+4);

Suppose the fib_exit = 0x81fc084C, pc = 0x81fc08124,

beq $s3,$s7,fib_exit

Relative addr = addr–(pc+4) =0x81fc084C–0x81fc08128 =0x24
Then rel addr>>2 = fib_exit >> 2 = 0x00000024 >> 2
= 0000 0000 0000 0000 0000 0000 0010 0100>>2
= 0000 0000 0000 0000 0000 0000 0000 1001
= 0x0000009

0x1fc08124    beq   $s3,$s7,fib_exit

000100: 00011 00111 0000000000001001
Executing Branch Instructions

Branch \[ \text{beq} \ \$rs,\$rt,\text{offset16} \]
\[ \$pc = (\$rt == \$rs)? (\$pc+4+(\delta^{32}(\text{offset16})<<2))): (\$pc+4); \]

Suppose the \( \text{pc} = 0x81fc08124 \),

\[ \text{beq} \ \$s3,\$s7,\text{fib}_\text{exit} \]

\[ \begin{array}{c}
000100: 00011 00111 00000000000001001 \\
\end{array} \]

Then address = 0x00000009
Then address \( << 2 \) = 0x00000024
Then \( \$pc+4 \) = 0x81fc08128
Then \( \$pc+4 + \text{address}<<2 \) = 0x81fc0814c

If branch occurred then \( \text{pc} \) = 0x81fc0814c
else \( \text{pc} \) = 0x81fc08128
Signed Binary numbers

Assume the word size is 4 bits,
Then each bit represents a power = \([-2^3]2^22^12^0 = S421\]

\(S\) represents the minus sign bit = \(-2^3 = -8\)

<table>
<thead>
<tr>
<th>S421</th>
<th>0000</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>S421</td>
<td>0001</td>
<td>1</td>
</tr>
<tr>
<td>S421</td>
<td>0010</td>
<td>2</td>
</tr>
<tr>
<td>S421</td>
<td>0011</td>
<td>3 = 2+1</td>
</tr>
<tr>
<td>S421</td>
<td>0100</td>
<td>4</td>
</tr>
<tr>
<td>S421</td>
<td>0101</td>
<td>5 = 4+1</td>
</tr>
<tr>
<td>S421</td>
<td>0110</td>
<td>6 = 4+2</td>
</tr>
<tr>
<td>S421</td>
<td>0111</td>
<td>7 = 4+2+1</td>
</tr>
<tr>
<td>S421</td>
<td>1000</td>
<td>-8 = -8+0</td>
</tr>
<tr>
<td>S421</td>
<td>1001</td>
<td>-7 = -8+1</td>
</tr>
<tr>
<td>S421</td>
<td>1010</td>
<td>-6 = -8+2</td>
</tr>
<tr>
<td>S421</td>
<td>1011</td>
<td>-5 = -8+2+1</td>
</tr>
<tr>
<td>S421</td>
<td>1100</td>
<td>-4 = -8+4</td>
</tr>
<tr>
<td>S421</td>
<td>1101</td>
<td>-3 = -8+4+1</td>
</tr>
<tr>
<td>S421</td>
<td>1110</td>
<td>-2 = -8+4+2</td>
</tr>
<tr>
<td>S421</td>
<td>1111</td>
<td>-1 = -8+4+2+1</td>
</tr>
</tbody>
</table>

unsigned 4 bit number: 0 to \(2^4 = 0..15\)
signed 4 bit number: \(-2^3 \) to \(2^3 - 1 = -8 .. 7\)

Sign numbers causes the loss of 1 bit accuracy
This is why C language provides signed & unsigned keywords
1’s and 2’s complement

One`s complement: invert each bit

For example: 0100 becomes 1011  (Note: 1011 is –5 and not –4)

The C language has a 1’s complement bitwise operator tilde (~).
  (i.e. ~1011 becomes 0100)

The 1’s complement operator has the property:  $X = ~~~X$;

Two’s complement number (also negation) is expressed as

  two’s complement $= -X = (~X)+1$

The 2’s complement operator has the property:  $X = - -X$;

  For example: 4 becomes –4

  For example: 0100 becomes $(1011+0001) = 1100 = -4$
Sign extension

Suppose we want to sign extend a 4-bit word to 8 bits.
Then take the value of the sign bit and propagate it.

For example: 1011 becomes 11111011

- Two’s complement allows the number to retain the same value even though we are adding 1’s!
  - $11111011 = -128 + 64 + 32 + 16 + 8 + 2 + 1 = -5$
  - $1011 = -8 + 2 + 1 = -5$

- Two’s complement allows us to treat the sign bit as another digit!
1-bit addition

The rules to add 1 bit numbers: Cin=Carry in; Cout=Carry Out

<table>
<thead>
<tr>
<th>Input</th>
<th>2^1</th>
<th>2^0</th>
<th>=2^1 2^0</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>Cin</td>
<td>Cout</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Sum = oddparity(A, B, Cin) = “odd number of bits”

Cout = majority(A, B, Cin) = “majority vote”

1-bit sum is the same as adding the bits modular 2 (i.e base 2).
N-bit addition

N-bit addition requires using only the 1-bit addition table

Suppose we want to add: 0101 + 0011 = 5 + 3 = 8

<table>
<thead>
<tr>
<th>Cin</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Sum</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Cout</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If the word size is a 4 bits then the Sum of 1000 is really -8 which is incorrect. Hence the number field is too small.

This is called arithmetic overflow = Cin_{sign} \^ Cout_{sign}

Is the exclusive-or of the Cin and the Cout of the sign bit field
N-bit subtraction

Two’s complement allows us to treat N-bit subtraction as N-bit addition.

Suppose we want to add: \( 5 - 3 = 0101 - 0011 = 3 \)

First 2’s complement 3: \( 0011 \Rightarrow 1100 + 1 \Rightarrow 1101 \)

Now just do addition: \( 5 + -3 = 0101 + 1101 \)

<table>
<thead>
<tr>
<th>Cin</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>-3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Sum</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Cout</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Overflow = 0

\[
\text{arithmetic overflow bit} = \text{Cin}_{\text{sign}} \wedge \text{Cout}_{\text{sign}} = 1 \wedge 1 = 0
\]
Multiply instruction

Two’s complement allows us to also multiply by addition

\[ 1 \times 1 = 1 \quad \text{and} \quad 0 \times M = 0 \]

Warning: for each sub-product, you must extend the sign bit

Note: a \( N \times N \) multiply results in a \( 2N \) product = \( 4 \times 4 = 8 \) bit

\[
\begin{array}{cccccccccccc}
-3 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\
+5 & & & & & & & & 0 & 1 & 0 & 1 \\
\hline \\
\multicolumn{12}{c}{1} \\
\end{array}
\]

\[
\begin{array}{cccccccccccc}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\
\hline \\
1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\
\end{array}
\]

Thus a \( 4 \times 4 \) multiply = 8 bit product. Add time = 1 clock.
N x N Multiply

Easier to place positive on top?

\[
\begin{array}{cccccc}
5 & 0 & 1 & 0 & 1 \\
-3 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\
\end{array}
\]

Add time = 6 clocks
MIPS multiply instruction

The MIPS does not have a general purpose multiply. It is required to copy the values in to special registers. Also, 32 x 32 multiply results in a 64 bit product.

In fact, some RISC machines, use only shift instructions and claim the same performance as machines with a hardware multiply!

Unsigned multiply: \texttt{multu} $rs,$rt # hi\_lo\_64 = $rs \times $rt

Signed multiply: \texttt{mult} $rs,$rt # hi\_lo\_64 = $rs \times $rt

move from low: \texttt{mflo} $rd # $rd = \text{lo}

move from high: \texttt{mfhi} $rd # $rd = \text{hi}

What is the MIPS for the following C code?

\begin{align*}
\text{int } x, y; & \quad y = 9*x + 2;
\end{align*}
Multiply by powers of 2: using shift

- The binary radix allows for easy multiply by powers of 2.
- The reason to use shifting is because it is fast (just move bits).
- In fact, many computers use a barrel shifter.
- A barrel shifter, shifts any amount in one clock cycle.
- Whereas a so-so multiplier may take up to \( n \) clocks.
- Multiply by a constant allows for further optimization.
- For example, \( x \times 9 = x \times (8 + 1) = x \times 8 + x \times 1 \)

\[
\begin{align*}
& \text{sll } \$s1,\$s0,3 \quad \# 8 = 2^3 \\
& \text{add } \$s1,\$s0,\$0 \quad \# x \times 9
\end{align*}
\]

What is the MIPS for the following C code?

```c
int x, y; y = 18*x + x/4;
```
Review: R-type instruction datapath (chapter 5)

R - Format

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>func</th>
</tr>
</thead>
</table>

ALU

func $rd, $rs, $rt

ALU control

Zero

ALU result

Read register 1
Read register 2
Write register
Write data

Read data 1
Read data 2

RegWrite

32

5

32

5

32

5

32

5
Review: Lw I-type instruction datapath

I - Format

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>offset</th>
</tr>
</thead>
</table>

Data Transfer

```
lw $rt, offset($rs)
```

```
op   rs   rt   offset
```
Review: Sw I-type instruction datapath

I - Format

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>offset</th>
</tr>
</thead>
</table>

Data Transfer: `sw $rt, offset($rs)`
Review: Branch I-type instruction datapath

I - Format

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>offset</th>
</tr>
</thead>
</table>

Branch instruction:

```
beq $rs,$rt,offset
```

ALU control

Sign extend

Shift left 2

Write register

Read data 1

Read data 2

RegWrite

ALU Zero

PC +4

ALU Zero

PC
### ALU decoder

<table>
<thead>
<tr>
<th>Machine opcode</th>
<th>Instruct Format</th>
<th>IR[31-26] Opcode</th>
<th>ALUop</th>
<th>IR[5-0]</th>
<th>ALUctl</th>
<th>ALUctl</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>I-type</td>
<td>100011</td>
<td>00</td>
<td>XXXXXX</td>
<td>add</td>
<td>010</td>
</tr>
<tr>
<td>sw</td>
<td>I-type</td>
<td>101011</td>
<td>00</td>
<td>XXXXXX</td>
<td>add</td>
<td>010</td>
</tr>
<tr>
<td>beq</td>
<td>I-type</td>
<td>000100</td>
<td>01</td>
<td>XXXXXX</td>
<td>sub</td>
<td>110</td>
</tr>
<tr>
<td>add</td>
<td>R-type</td>
<td>000000</td>
<td>10</td>
<td>100000</td>
<td>add</td>
<td>010</td>
</tr>
<tr>
<td>sub</td>
<td>R-type</td>
<td>000000</td>
<td>10</td>
<td>100010</td>
<td>sub</td>
<td>110</td>
</tr>
<tr>
<td>and</td>
<td>R-type</td>
<td>000000</td>
<td>10</td>
<td>100100</td>
<td>and</td>
<td>000</td>
</tr>
<tr>
<td>or</td>
<td>R-type</td>
<td>000000</td>
<td>10</td>
<td>100101</td>
<td>or</td>
<td>001</td>
</tr>
<tr>
<td>slt</td>
<td>R-type</td>
<td>000000</td>
<td>10</td>
<td>101010</td>
<td>slt</td>
<td>111</td>
</tr>
</tbody>
</table>

\[
\text{ALUop} = \text{ALUopDecoder}(\text{Opcode});
\]

\[
\text{ALUctl} = \text{ALUctlDecoder}(\text{ALUop}, \text{Funct});
\]

Note: the Opcode field determines the I-type and R-type

Note: the Funct field determines the ALUctl for R-type
ALU decoders: ALUop and ALUctl

<table>
<thead>
<tr>
<th>op</th>
<th>IR[31-26]</th>
<th>ALUop</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>100011</td>
<td>00</td>
</tr>
<tr>
<td>sw</td>
<td>101011</td>
<td>00</td>
</tr>
<tr>
<td>beq</td>
<td>000100</td>
<td>01</td>
</tr>
<tr>
<td>add</td>
<td>000000</td>
<td>10</td>
</tr>
<tr>
<td>sub</td>
<td>000000</td>
<td>10</td>
</tr>
<tr>
<td>and</td>
<td>000000</td>
<td>10</td>
</tr>
<tr>
<td>or</td>
<td>000000</td>
<td>10</td>
</tr>
<tr>
<td>slt</td>
<td>000000</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>op</th>
<th>IR[5-0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>xxxxxxxx</td>
</tr>
<tr>
<td>sw</td>
<td>xxxxxxxx</td>
</tr>
<tr>
<td>beq</td>
<td>xxxxxxxx</td>
</tr>
<tr>
<td>add</td>
<td>100000</td>
</tr>
<tr>
<td>sub</td>
<td>100010</td>
</tr>
<tr>
<td>and</td>
<td>100100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ALUctl</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>bitwise and</td>
</tr>
<tr>
<td>001</td>
<td>bitwise or</td>
</tr>
<tr>
<td>010</td>
<td>integer add</td>
</tr>
<tr>
<td>110</td>
<td>integer sub</td>
</tr>
<tr>
<td>111</td>
<td>set less than</td>
</tr>
</tbody>
</table>

ALUop

31-26

ALU ctl

32

ALU control

32
Processor architecture with ALU decoder

 Opcode: IR[31-26]

 Funct: IR[5-0]
R-format datapath control

(Figures 5.20-24)

<table>
<thead>
<tr>
<th>Machine</th>
<th>opcode</th>
<th>R-format</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>Memto</th>
<th>Reg</th>
<th>Write</th>
<th>Mem</th>
<th>Mem</th>
<th>Mem</th>
<th>Branch</th>
<th>ALUop</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R-format</td>
<td>1 ($rd)</td>
<td>0 ($rt)</td>
<td>0(alu)</td>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
</tr>
</tbody>
</table>

![Diagram of R-format datapath control](image_url)
lw datapath control

(Figure 5.25)

<table>
<thead>
<tr>
<th>Machine</th>
<th>opcode</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>Memto</th>
<th>Reg</th>
<th>Write</th>
<th>Read</th>
<th>Mem</th>
<th>Write</th>
<th>Branch</th>
<th>ALUop</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>0</td>
<td>($rt)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>01</td>
</tr>
</tbody>
</table>

[Diagram showing the lw datapath control with instructions and data flow.]
sw datapath control

<table>
<thead>
<tr>
<th>Machine</th>
<th>opcode</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>Memto</th>
<th>Reg Write</th>
<th>Mem Read</th>
<th>Mem Write</th>
<th>Branch</th>
<th>ALUop</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>sw</td>
<td>X</td>
<td>1 (offset)</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>01 (add)</td>
</tr>
</tbody>
</table>

![Diagram of SW Datapath Control]
beq datapath control

(Figure 5.26)

<table>
<thead>
<tr>
<th>Machine</th>
<th>opcode</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>Memto</th>
<th>Reg</th>
<th>Write</th>
<th>Mem Read</th>
<th>Mem Write</th>
<th>Branch</th>
<th>ALUop</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>beq</td>
<td>X</td>
<td>0</td>
<td></td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>01 (sub)</td>
</tr>
</tbody>
</table>

Diagram of the beq instruction's datapath control.
A multi-cycle processor has the following instruction times:

- **add (44%)** = 6ns = Fetch(2ns) + RegR(1ns) + ALU(2ns) + RegW(1ns)
- **lw (24%)** = 8ns = Fetch(2ns) + RegR(1ns) + ALU(2ns) + MemR(2ns) + RegW(1ns)
- **sw (12%)** = 7ns = Fetch(2ns) + RegR(1ns) + ALU(2ns) + MemW(2ns)
- **beq (18%)** = 5ns = Fetch(2ns) + RegR(1ns) + ALU(2ns)
- **j (2%)** = 2ns = Fetch(2ns)

Single-cycle CPI = 44%×8ns + 24%×8ns + 12%×8ns + 18%×8ns + 2%×8ns = 8ns
Multi-cycle CPI = 44%×6ns + 24%×8ns + 12%×7ns + 18%×5ns + 2%×2ns = 6.3ns

\[
\frac{\text{single - cycle CPI}}{\text{multi - cycle CPI}} = \frac{8ns}{6.3ns} = 1.27 \text{ times faster}
\]

**Architectural improved performance without speeding up the clock!**
Single-cycle problems

- **Single Cycle Problems:**
  - Clock cycle is the slowest instruction delay = 8ns = 125MHz
  - What if we had a more complicated instruction like floating point? (fadd = 30ns, fmul=100ns) Then clock cycle = 100ns = 10 MHz
  - Wasteful of chip area (2 adders + 1 ALU). Cannot reuse resources.
  - Wasteful of memory: separate instructions & data (Harvard architecture)

- **Solutions:**
  - Use a “smaller” cycle time (if the technology can do it)
  - Have different instructions take different numbers of cycles (multi-cycle)
  - Better reuse of functional units: a “multicycle” datapath (1 ALU instead of 3 adders)

- **Multi-cycle approach**
  - Clock cycle is the slowest function unit = 2ns = 500MHz
  - We will be reusing functional units:
    - ALU used to increment PC (Adder1) & compute address (Adder2)
  - Memory reused for instruction and data (Von Neuman architecture)
Some Design Trade-offs

High level design techniques

Algorithms: change instruction usage

\[ \text{minimize } \sum n_{\text{instruction}} \times t_{\text{instruction}} \]

Architecture: Datapath, FSM, Microprogramming
adders: ripple versus carry lookahead
multiplier types, …

Lower level design techniques (closer to physical design)
clocking: single versus multi clock
technology: layout tools: better place and route
process technology: 0.5 micron to .18 micron
Multi-cycle Datapath: with controller
Multi-cycle Datapath

Multi-cycle = 1 Mem + 5.5 Muxes + 1 ALU + 5 Registers (A,B,IR,MDR,ALUOut)

Single-cycle = 2 Mem + 4.0 Muxes + 1 ALU + 2 adders
Multi-cycle: 5 execution steps

- $T_1 (a, lw, sw, beq, j)$ Instruction Fetch
- $T_2 (a, lw, sw, beq, j)$ Instruction Decode and Register Fetch
- $T_3 (a, lw, sw, beq, j)$ Execution, Memory Address Calculation, or Branch Completion
- $T_4 (a, lw, sw)$ Memory Access or R-type instruction completion
- $T_5 (a, lw)$ Write-back step

**INSTRUCTIONS TAKE FROM 3 - 5 CYCLES!**
# Multi-cycle Approach

All operations in each clock cycle $T_i$ are done in parallel not sequential!
For example, $T_1$, IR = Memory[PC] and PC=PC+4 are done simultaneously!

<table>
<thead>
<tr>
<th>Step name</th>
<th>Action for R-type instructions</th>
<th>Action for memory-reference instructions</th>
<th>Action for branches</th>
<th>Action for jumps</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$ Instruction fetch</td>
<td></td>
<td>IR = Memory[PC]</td>
<td>PC = PC + 4</td>
<td></td>
</tr>
<tr>
<td>$T_2$ Instruction decode/ register fetch</td>
<td></td>
<td>A = Reg [IR[25-21]]</td>
<td>B = Reg [IR[20-16]]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ALUOut = PC + (sign-extend (IR[15-0]) &lt;&lt; 2)</td>
<td></td>
</tr>
<tr>
<td>$T_3$ Execution, address computation, branch/</td>
<td>ALUOut = A op B</td>
<td>ALUOut = A + sign-extend (IR[15-0])</td>
<td>if (A == B) then</td>
<td></td>
</tr>
<tr>
<td>jump completion</td>
<td></td>
<td></td>
<td>PC = ALUOut</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>PC = PC [31-28] II</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(IR[25-0]&lt;&lt;2)</td>
<td></td>
</tr>
<tr>
<td>$T_4$ Memory access or R-type completion</td>
<td>Reg [IR[15-11]] = ALUOut</td>
<td>Load: MDR = Memory[ALUOut] or Store: Memory [ALUOut] = B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$T_5$ Memory read completion</td>
<td></td>
<td>Load: Reg[IR[20-16]] = MDR</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Between Clock $T_2$ and $T_3$ the microcode sequencer will do a dispatch 1
Multi-cycle using Microprogramming

Finite State Machine (hardwired control)

- Inputs from instruction register opcode field
- State register
- Outputs from combinational control logic
- Datapath control outputs
- Next state

Microcode controller

- Microcode storage
- Outputs
- Datapath control outputs
- Input
- Sequencing control
- Adder
- Address select logic
- Microprogram counter
- Inputs from instruction register opcode field

Requires microcode memory to be faster than main memory
Microcode: Trade-offs

- Distinction between specification & implementation is sometimes blurred

- Specification Advantages:
  - Easy to design and write (maintenance)
  - Design architecture and microcode in parallel

- Implementation (off-chip ROM) Advantages
  - Easy to change since values are in memory
  - Can emulate other architectures
  - Can make use of internal registers

- Implementation Disadvantages, SLOWER now that:
  - Control is implemented on same chip as processor
  - ROM is no longer faster than RAM
  - No need to go back and make changes
## Microinstruction format

<table>
<thead>
<tr>
<th>Field name</th>
<th>Value</th>
<th>Signals active</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ALU control</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td>ALUOp = 00</td>
<td></td>
<td>Cause the ALU to add.</td>
</tr>
<tr>
<td>Subt</td>
<td>ALUOp = 01</td>
<td></td>
<td>Cause the ALU to subtract; this implements the compare for branches.</td>
</tr>
<tr>
<td>Func code</td>
<td>ALUOp = 10</td>
<td></td>
<td>Use the instruction’s function code to determine ALU control.</td>
</tr>
<tr>
<td><strong>SRC1</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PC</td>
<td>ALUSrcA = 0</td>
<td></td>
<td>Use the PC as the first ALU input.</td>
</tr>
<tr>
<td>A</td>
<td>ALUSrcA = 1</td>
<td></td>
<td>Register A is the first ALU input.</td>
</tr>
<tr>
<td><strong>SRC2</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>ALUSrcB = 00</td>
<td></td>
<td>Register B is the second ALU input.</td>
</tr>
<tr>
<td>4</td>
<td>ALUSrcB = 01</td>
<td></td>
<td>Use 4 as the second ALU input.</td>
</tr>
<tr>
<td>Extend</td>
<td>ALUSrcB = 10</td>
<td></td>
<td>Use output of the sign extension unit as the second ALU input.</td>
</tr>
<tr>
<td>Extshift</td>
<td>ALUSrcB = 11</td>
<td></td>
<td>Use the output of the shift-by-two unit as the second ALU input.</td>
</tr>
<tr>
<td><strong>Register control</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read</td>
<td></td>
<td></td>
<td>Read two registers using the rs and rt fields of the IR as the register numbers and putting the data into registers A and B.</td>
</tr>
<tr>
<td>Write ALU</td>
<td>RegWrite,</td>
<td>RegDst = 1, MemtoReg = 0</td>
<td>Write a register using the rd field of the IR as the register number and the contents of the ALUOut as the data.</td>
</tr>
<tr>
<td>Write MDR</td>
<td>RegWrite,</td>
<td>RegDst = 0, MemtoReg = 1</td>
<td>Write a register using the rt field of the IR as the register number and the contents of the MDR as the data.</td>
</tr>
<tr>
<td><strong>Memory</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read PC</td>
<td>MemRead,</td>
<td>lorD = 0</td>
<td>Read memory using the PC as address; write result into IR (and the MDR).</td>
</tr>
<tr>
<td>Read ALU</td>
<td>MemRead,</td>
<td>lorD = 1</td>
<td>Read memory using the ALUOut as address; write result into MDR.</td>
</tr>
<tr>
<td>Write ALU</td>
<td>MemWrite,</td>
<td>lorD = 1</td>
<td>Write memory using the ALUOut as address, contents of B as the data.</td>
</tr>
<tr>
<td><strong>PC write control</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALU</td>
<td>PCSource = 00</td>
<td></td>
<td>Write the output of the ALU into the PC.</td>
</tr>
<tr>
<td>ALUOut-cond</td>
<td>PCSource = 01</td>
<td></td>
<td>If the Zero output of the ALU is active, write the PC with the contents of the register ALUOut.</td>
</tr>
<tr>
<td>jump address</td>
<td>PCSource = 10</td>
<td></td>
<td>Write the PC with the jump address from the instruction.</td>
</tr>
<tr>
<td><strong>Sequencing</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Seq</td>
<td>AddrCtl = 11</td>
<td></td>
<td>Choose the next microinstruction sequentially.</td>
</tr>
<tr>
<td>Fetch</td>
<td>AddrCtl = 00</td>
<td></td>
<td>Go to the first microinstruction to begin a new instruction.</td>
</tr>
<tr>
<td>Dispatch 1</td>
<td>AddrCtl = 01</td>
<td></td>
<td>Dispatch using the ROM 1.</td>
</tr>
<tr>
<td>Dispatch 2</td>
<td>AddrCtl = 10</td>
<td></td>
<td>Dispatch using the ROM 2.</td>
</tr>
</tbody>
</table>
Microinstruction format: Maximally vs. Minimally Encoded

• **No encoding:**
  – 1 bit for each datapath operation
  – faster, requires more memory (logic)
  – used for Vax 780 — an astonishing 400K of memory!

• **Lots of encoding:**
  – send the microinstructions through logic to get control signals
  – uses less memory, slower

• **Historical context of CISC:**
  – Too much logic to put on a single chip with everything else
  – Use a ROM (or even RAM) to hold the microcode
  – It’s easy to add new instructions
**Microprogramming: program**

<table>
<thead>
<tr>
<th>Label</th>
<th>ALU control</th>
<th>SRC1</th>
<th>SRC2</th>
<th>Register control</th>
<th>Memory</th>
<th>PCWrite control</th>
<th>Sequencing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>Add</td>
<td>PC</td>
<td>4</td>
<td>Read PC</td>
<td>ALU</td>
<td>Seq</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Add</td>
<td>PC</td>
<td>Extshft</td>
<td>Read</td>
<td></td>
<td>Dispatch 1</td>
<td></td>
</tr>
<tr>
<td>Mem1</td>
<td>Add</td>
<td>A</td>
<td>Extend</td>
<td></td>
<td></td>
<td>Dispatch 2</td>
<td></td>
</tr>
<tr>
<td>LW2</td>
<td>Add</td>
<td>A</td>
<td>Extend</td>
<td>Read ALU</td>
<td>Seq</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SW2</td>
<td></td>
<td></td>
<td></td>
<td>Write MDR</td>
<td>Fetch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rformat1</td>
<td>Func code</td>
<td>A</td>
<td>B</td>
<td>Write ALU</td>
<td>Seq</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Fetch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BEQ1</td>
<td>Subt</td>
<td>A</td>
<td>B</td>
<td>ALUOut-cond</td>
<td>Fetch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JUMP1</td>
<td></td>
<td></td>
<td></td>
<td>Jump address</td>
<td>Fetch</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Step name</strong></th>
<th><strong>Action for R-type instructions</strong></th>
<th><strong>Action for memory-reference instructions</strong></th>
<th><strong>Action for branches</strong></th>
<th><strong>Action for jumps</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction fetch</td>
<td>IR = Memory[PC]</td>
<td></td>
<td>PC = PC + 4</td>
<td></td>
</tr>
<tr>
<td>Instruction decode/register fetch</td>
<td></td>
<td>A = Reg [IR[25-21]]</td>
<td>B = Reg [IR[20-16]]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ALUOut = PC + (sign-extend (IR[15-0]) &lt;&lt; 2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execution, address computation, branch/</td>
<td>ALUOut = A op B</td>
<td>ALUOut = A + sign-extend (IR[15-0])</td>
<td>if (A == B) then</td>
<td>PC = PC [31-28] II</td>
</tr>
<tr>
<td>jump completion</td>
<td></td>
<td></td>
<td>PC = ALUOut</td>
<td>(IR[25-0]&lt;&lt;&lt;2)</td>
</tr>
<tr>
<td>Memory access or R-type completion</td>
<td>Reg [IR[15-11]] = ALUOut</td>
<td>Load: MDR = Memory[ALUOut] or</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Store: Memory [ALUOut] = B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory read completion</td>
<td>Load: Reg[IR[20-16]] = MDR</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Microprogramming: program overview

Fetch

Fetch+1

Dispatch 1

T1

Rformat1

BEQ1

JUMP1

Mem1

LW2

SW2

T2

T3

T4

Rformat1+1

T5

LW2+1
Microprogram stepping: \( T_1 \) Fetch

(Done in parallel) \( IR \leftarrow \text{MEMORY}[PC] \) & \( PC \leftarrow PC + 4 \)
\[ T_2 \text{ Fetch } + 1 \]

\[ A \leftarrow \text{Reg[IR[25-21]]} \land B \leftarrow \text{Reg[IR[20-16]]} \land \text{ALUOut} \leftarrow \text{PC+signext(IR[15-0])} \ll 2 \]
\[ \text{ALUOut} \leftarrow A + \text{sign\_extend}([IR[15-0]]) \]
Dispatch 2: LW2

MDR ← Memory[ALUOut]
Reg[ IR[20-16] ] ← MDR
Dispatch 2: SW2

Memory[ ALUOut ] ← B

Label ALU SRC1 SRC2 RCntl Memory WriteALU PCwrite Seq Fetch
SW2

CWRU EECS 322  44
T₃ Dispatch 1: Rformat1

\[ \text{ALUOut} \leftarrow A \ \text{op}(\text{IR}[31-26]) \ B \]

**Diagram:**
- **PC**
- **0 Mux 1**
- **Address Memory MemData**
- **Write data**
- **Instruction [25–21]**
- **Instruction [20–16]**
- **Instruction [15–11]**
- **Instruction [15–0]**
- **Memory data register**
- **0 Mux 1**
- **0 Mux 1**
- **Instruction [15–0]**
- **Instruction [5–0]**
- **MemtoReg**
- **ALU result**
- **ALU OUT**
- **ALU control**
- **Zero**
- **ALUSrcB ALUOp**

**Tables:**
- **Label**: Rf...1
- **ALU op**: A
- **SRC1**: A
- **SRC2**: B
- **RCntl**: op(IR[31-26])
- **Memory**: Seq
- **PCwrite**: Seq
- **Seq**: Seq
**Dispatch 1: Rformat1+1**


Diagram showing the flow of data and control signals through various modules such as PC, ALU, Instruction Register (IR), Memory, and registries. The diagram illustrates the process of fetching, decoding, and executing instructions with specific paths for addressing, data read/write, and control signals.

### Labels and Symbols
- **Label**: Instructions or control signals.
- **ALU**: Arithmetic Logic Unit.
- **SRC1**: Source register 1.
- **SRC2**: Source register 2.
- **RCntl**: Register control.
- **Memory**: Memory access.
- **IR**: Instruction Register.
- **WALU**: Word ALU.
- **PCwrite**: Program Counter write.
- **Seg**: Segment.
- **Fetch**: Instruction fetch.

### Key Concepts
- **Instruction Decode**: The process of interpreting the instruction fields to determine the operation and operands.
- **ALU Operations**: Include arithmetic and logic operations like add, subtract, logical AND, OR, etc.
- **Memory Access**: Reading and writing data to memory for instructions and intermediate results.
- **Register Operations**: Read, write, and transfer data between registers.

### DiagramDetails
- **PC (Program Counter)**: Batches the program counter with a multiplexer (MUX) to select the next address.
- **Address**: Calculates the memory address based on the program counter.
- **Memory**: Accesses memory data, which can be read or written.
- **ALU**: Performs arithmetic and logic operations based on the input data and control signals.
- **Instruction**: Represents the current instruction being processed.
- **Registers**: Contain data used for intermediate calculations and final results.
- **Sign Extend**: Extends the sign bit of the input data for certain operations.
- **Shift**: Performs left shift operations on data.
- **ALU Out**: The output of the ALU, which can be directed to various destinations.

The diagram highlights the flow of data and control signals, showcasing how each component interacts to execute instructions efficiently.
If \((A - B == 0)\) \{ \text{PC} \leftarrow \text{ALUOut}; \}
$T_3$ Dispatch 1: Jump1

$PC \leftarrow PC[31-28] \ || \ IR[25-0] \ll< 2$

---

**Label**  ALU  SRC1  SRC2  RCntl  Memory  PCwrite  Seq  Fetch
Jump1