Lecture 7
Designing a Single Cycle Datapath

September 20, 1999
John Kubiatowicz (http.cs.berkeley.edu/~kubitron)

lecture slides: http://www-inst.eecs.berkeley.edu/~cs152/

Outline of Today’s Lecture
° Recap (5 minutes)
° Finish on Floating Point
° Design a processor: step-by-step
° Requirements of the Instruction Set
° Questions and Administrative Matters (5 minutes)
° Components and Clocking
° Assembling an Adequate Datapath
° Break (5 minutes)
° Controlling the datapath

Review: DIVIDE HARDWARE Version 3
° 32-bit Divisor reg, 32-bit ALU, 64-bit Remainder reg, (0-bit Quotient reg)

° Multiplication and Division can use same hardware!

Review: IEEE Floating-Point
Representation of floating point numbers in IEEE 754 standard:
single precision

<table>
<thead>
<tr>
<th>sign</th>
<th>E</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>8</td>
<td>23</td>
</tr>
</tbody>
</table>

actual exponent is e = E - 127

-0 < E < 255

0 = 00000000 0 . . . 0
-1.5 = 1 01111111 10 . . . 0

Magnitude of numbers that can be represented is in the range:

-2^{-126} (1.0) to 2^{127} (2 - 2^{23})

which is approximately:

1.8 x 10^{-38} to 3.40 x 10^{38}

(integer comparison valid on IEEE Fl.Pt. numbers of same sign!)
Review: Floating Point rounding/computations

- Bits have no inherent meaning; it is the context which determines whether they are ASCII characters, integers, floating point numbers, or instructions
- IEEE Standard: four rounding modes:
  - round to nearest even (default)
  - round towards plus infinity
  - round towards minus infinity
  - round towards zero

round to nearest:
  round digit < B/2 then truncate
  > B/2 then round up (add 1 to ULP: unit in last place)
  = B/2 then round to nearest even digit

- Arithmetic algorithms just as in high school
  - Addition:
    - shift number with smaller exponent right till have same scale
    - add mantissas
    - renormalize
  - Subtraction? Multiplication? Division?

Review: Extra Bits for rounding

"Floating Point numbers are like piles of sand; every time you move one you lose a little sand, but you pick up a little dirt."

How many extra bits?
IEEE: As if computed the result exactly and rounded.
Addition:
- 1.xxxxx 1.xxxxx 1.xxxxx
- + 1.xxxxx 0.001xxxxx 0.01xxxxx
- 1x.xxxxxy 1.xxxxxxyy 1x.xxxxxyy

post-normalization pre-normalization pre and post

- Guard Digits: digits to the right of the first p digits of significand to guard against loss of digits – can later be shifted left into first P places during normalization.
- Addition: carry-out shifted in
- Subtraction: borrow digit and guard
- Multiplication: carry and guard, Division requires guard

Sticky Bit

Additional bit to the right of the round digit to better fine tune rounding

\[ d_0 \cdot d_1 d_2 d_3 \ldots dp-1 \ 0 \ 0 \ 0 \ + \ 0 \ 0 \ 0 \ x \ x \ \ldots \ x \ x \ \ldots \ ]

Sticky bit: set to 1 if any 1 bits fall off the end of the round digit

\[ d_0 \cdot d_1 d_2 d_3 \ldots dp-1 \ 0 \ 0 \ 0 \ - \ 0 \ 0 \ 0 \ x \ x \ \ldots \ x \ x \ \ldots \ ]

- generates a borrow

Rounding Summary:
Radix 2 minimizes wobble in precision

Normal operations in +,-,*,/ require one carry/borrow bit + one guard digit

One round digit needed for correct rounding

Sticky bit needed when round digit is B/2 for max accuracy

Rounding to nearest has mean error = 0 if uniform distribution of digits are assumed

Denormalized Numbers

\[ \frac{2^{-22}}{2^{2-22}} \]

\[ \frac{2^{-22}}{2^{2-22}} \]

The gap between 0 and the next representable number is much larger than the gaps between nearby representable numbers.

IEEE standard uses denormalized numbers to fill in the gap, making the distances between numbers near 0 more alike.

\[ \frac{2^{-22}}{2^{2-22}} \]

p-bits of precision

p bits of precision

same spacing, half as many values!

NOTE: PDP-11, VAX cannot represent subnormal numbers. These machines underflow to zero instead.
**Infinity and NaNs**

result of operation *overflows*, i.e., is larger than the largest number that can be represented

overflow is not the same as divide by zero (raises a different exception)

\[ +/- \infty \begin{array}{c} 1 \ldots 1 \nonumber \\
\ldots 0 \ldots 0 \end{array} \]

It may make sense to do further computations with infinity 

\[ e.g., \frac{X}{0} > Y \text{ may be a valid comparison} \]

Not a number, but not infinity (e.g. \( \sqrt{-4} \))

\[ \text{invalid operation exception (unless operation is = or \neq)} \]

\[ \text{NaN} \begin{array}{c} 1 \ldots 1 \text{ non-zero} \\
\ldots \end{array} \]

HW decides what goes here

NaNs propagate: \( f(NaN) = NaN \)

**Radix-4 Modified Booth’s => Multiple representations**

Once admit new symbols (i.e. \( \overline{1} \)), can have multiple representations of a number:

<table>
<thead>
<tr>
<th>Current Bits</th>
<th>Bit to the Right</th>
<th>Explanation</th>
<th>Example</th>
<th>Recode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>0</td>
<td>Middle of zeros</td>
<td>00 00 00</td>
<td>00 (0)</td>
</tr>
<tr>
<td>0 1</td>
<td>0</td>
<td>Single one</td>
<td>00 00 00</td>
<td>01 (1)</td>
</tr>
<tr>
<td>1 0</td>
<td>0</td>
<td>Begins run of 1s</td>
<td>00 01 11</td>
<td>01 (2)</td>
</tr>
<tr>
<td>1 1</td>
<td>0</td>
<td>Begins run of 1s</td>
<td>00 01 11</td>
<td>01 (2)</td>
</tr>
<tr>
<td>0 0</td>
<td>1</td>
<td>Ends run of 1s</td>
<td>00 00 11</td>
<td>00 (0)</td>
</tr>
<tr>
<td>0 1</td>
<td>1</td>
<td>Ends run of 1s</td>
<td>00 01 11</td>
<td>00 (0)</td>
</tr>
<tr>
<td>1 0</td>
<td>1</td>
<td>Isolated 0</td>
<td>00 11 10</td>
<td>00 (0)</td>
</tr>
<tr>
<td>1 1</td>
<td>1</td>
<td>Middle of run</td>
<td>00 11 11</td>
<td>00 (0)</td>
</tr>
</tbody>
</table>

**Pentium Bug**

* Pentium FP Divider uses algorithm to generate multiple bits per steps
  * FPU uses most significant bits of divisor & dividend/remainder to guess next 2 bits of quotient
  * Guess is taken from lookup table: -2, -1,0,+1,+2 (if previous guess too large a reminder, quotient is adjusted in subsequent pass of -2)
  * Guess is multiplied by divisor and subtracted from remainder to generate a new remainder
  * Called SRT division after 3 people who came up with idea

* Pentium table uses 7 bits of remainder + 4 bits of divisor = \( 2^{11} \) entries

* 5 entries of divisors omitted: 1.0001, 1.0100, 1.0111, 1.1010, 1.1101 from PLA (fix is just add 5 entries back into PLA: cost \$200,000)

* Self correcting nature of SRT \( \Rightarrow \) string of 1s must follow error
  * e.g. 1011 1111 1111 1111 1111 1011 1000 0010 0011 0111 1011 0100 (2,999999892918)

* Since indexed also by divisor/remainder bits, sometimes bug doesn’t show even with dangerous divisor value

**Pentium bug appearance**

* First 11 bits to right of decimal point always correct: bits 12 to 52 where bug can occur (4th to 15th decimal digits)

* FP divisors near integers 3, 9, 15, 21, 27 are dangerous ones:
  * \( 3.0 > d \geq 3.0 - 36 \times 2^{-22} \), \( 9.0 > d \geq 9.0 - 36 \times 2^{-20} \)
  * \( 15.0 > d \geq 15.0 - 36 \times 2^{-19} \), \( 21.0 > d \geq 21.0 - 36 \times 2^{-18} \)

* 0.333333 x 9 could be problem

* In Microsoft Excel, try \((4,195,835 / 3,145,727) \times 3,145,727\) \* 3,145,727
  * = 4,195,835 \( \Rightarrow \) not a Pentium with bug
  * = 4,195,579 \( \Rightarrow \) Pentium with bug
    (assuming Excel doesn’t already have SW bug patch)
  * Rarely noticed since error in 5th significant digit
  * Success of IEEE standard made discovery possible:
    - all computers should get same answer
Pentium Bug Time line

° June 1994: Intel discovers bug in Pentium: takes months to make change, reverify, put into production: plans good chips in January 1995 4 to 5 million Pentiums produced with bug
° Scientist suspects errors and posts on Internet in September 1994
° Nov. 22 Intel Press release: “Can make errors in 9th digit ... Most engineers and financial analysts need only 4 of 5 digits. Theoretical mathematician should be concerned. ... So far only heard from one.”
° Intel claims happens once in 27,000 years for typical spread sheet user:
  • 1000 divides/day x error rate assuming numbers random
° Dec 12: IBM claims happens once per 24 days: Bans Pentium sales
  • 5000 divides/second x 15 minutes = 4,200,000 divides/day
  • IBM statement: http://www.ibm.com/Features/pentium.html
  • Intel said it regards IBM’s decision to halt shipments of its Pentium processor-based systems as unwarranted.

Pentium conclusion: Dec. 21, 1994 $500M write-off

“To owners of Pentium processor-based computers and the PC community:

We at Intel wish to sincerely apologize for our handling of the recently publicized Pentium processor flaw.

The Intel Inside symbol means that your computer has a microprocessor second to none in quality and performance. Thousands of Intel employees work very hard to ensure that this is true. But no microprocessor is ever perfect.

What Intel continues to believe is technically an extremely minor problem has taken on a life of its own. Although Intel firmly stands behind the quality of the current version of the Pentium processor, we recognize that many users have concerns.

We want to resolve these concerns.

Intel will exchange the current version of the Pentium processor for an updated version, in which this floating-point divide flaw is corrected, for any owner who requests it, free of charge anytime during the life of their computer. Just call 1-800-628-8888.”

Sincerely,
Andrew S. Grove
President /CEO
Craig R. Barrett
Executive Vice President &COO
Gordon E. Moore
Chairman of the Board

The Big Picture: Where are We Now?

° The Five Classic Components of a Computer

° Today’s Topic: Design a Single Cycle Processor

° The Big Picture: The Performance Perspective

° Performance of a machine is determined by: CPI

  • Instruction count
  • Clock cycle time
  • Clock cycles per instruction

° Processor design (datapath and control) will determine:

  • Clock cycle time
  • Clock cycles per instruction

° Today:

  • Single cycle processor:
    - Advantage: One clock cycle per instruction
    - Disadvantage: long cycle time
Questions and Administrative Matters (5 Minutes)

° Reading Assignment 5.1-5.4
° Project teams -- choose next Tuesday:
  • Form four or five people project team.
  • We want you to learn to work in a big team.
  • Other project members must be in same section
° Make sure to look for assignments on Handouts page
° Midterm Wednesday 10/6 in 277 Cory 5:30PM-8:30PM
  • you may bring one double-sided page of notes
  • we’ll give you the opcode table from the book
  • review session Sunday before(?)
  • previous midterms and solutions on-line for review

How to Design a Processor: step-by-step

° 1. Analyze instruction set => datapath requirements
  • the meaning of each instruction is given by the register transfers
  • datapath must include storage element for ISA registers
    - possibly more
  • datapath must support each register transfer
° 2. Select set of datapath components and establish clocking methodology
° 3. Assemble datapath meeting the requirements
° 4. Analyze implementation of each instruction to determine setting of control points that effects the register transfer.
° 5. Assemble the control logic

The MIPS Instruction Formats

° All MIPS instructions are 32 bits long. The three instruction formats:

  \[
  \begin{array}{cccccccc}
  31 & 26 & 21 & 16 & 11 & 6 & 0 \\
  \hline
  \text{R-type} & \text{op} & \text{rs} & \text{rt} & \text{rd} & \text{shamt} & \text{funct} \\
  & 6 \text{ bits} & 5 \text{ bits} & 5 \text{ bits} & 5 \text{ bits} & 5 \text{ bits} & 5 \text{ bits} \\
  \text{I-type} & \text{op} & \text{rs} & \text{rt} & \text{immediate} \\
  & 6 \text{ bits} & 5 \text{ bits} & 5 \text{ bits} & 16 \text{ bits} \\
  \text{J-type} & \text{op} & \text{target address} \\
  & 6 \text{ bits} & 26 \text{ bits} \\
  \end{array}
  \]

° The different fields are:
  • op: operation of the instruction
  • rs, rt, rd: the source and destination register specifiers
  • shamt: shift amount
  • funct: selects the variant of the operation in the “op” field
  • address / immediate: address offset or immediate value
  • target address: target address of the jump instruction

Step 1a: The MIPS-lite Subset for today

° ADD and SUB
  • addU rd, rs, rt
  • subU rd, rs, rt

° OR Immediate:
  • ori rt, rs, imm16

° LOAD and STORE Word
  • lw rt, rs, imm16
  • sw rt, rs, imm16

° BRANCH:
  • beq rs, rt, imm16
Logical Register Transfers

- RTL gives the meaning of the instructions
- All start by fetching the instruction

\[
\begin{align*}
\text{op} & | \text{rs} | \text{rt} | \text{rd} | \text{shamt} | \text{funct} = \text{MEM}[\text{PC}] \\
\text{op} & | \text{rs} | \text{rt} | \text{Imm16} = \text{MEM}[\text{PC}] \\
\end{align*}
\]

<table>
<thead>
<tr>
<th>inst</th>
<th>Register Transfers</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDU</td>
<td>R[rd] ← R[rs] + R[rt]; PC ← PC + 4</td>
</tr>
<tr>
<td>SUBU</td>
<td>R[rd] ← R[rs] - R[rt]; PC ← PC + 4</td>
</tr>
<tr>
<td>ORi</td>
<td>R[rt] ← R[rs]</td>
</tr>
<tr>
<td>LOAD</td>
<td>R[rt] ← MEM[ R[rs] + sign_ext(Imm16)]; PC ← PC + 4</td>
</tr>
<tr>
<td>STORE</td>
<td>MEM[ R[rs] + sign_ext(Imm16) ] ← R[rt]; PC ← PC + 4</td>
</tr>
<tr>
<td>BEQ</td>
<td>if ( R[rs] == R[rt] ) then PC ← PC + 4 + sign_ext(Imm16)</td>
</tr>
</tbody>
</table>

Step 1: Requirements of the Instruction Set

- Memory
  - instruction & data
- Registers (32 x 32)
  - read RS
  - read RT
  - Write RT or RD
- PC
- Extender
  - Add and Sub register or extended immediate
  - Add 4 or extended immediate to PC

Step 2: Components of the Datapath

- Combinational Elements
- Storage Elements
  - Clocking methodology

Combinational Logic Elements (Basic Building Blocks)

- Adder
- MUX
- ALU
### Storage Element: Register (Basic Building Block)

- **Register**
  - Similar to the D Flip Flop except
    - N-bit input and output
    - Write Enable input
  - **Write Enable**:
    - Negated (0): Data Out will not change
    - Asserted (1): Data Out will become Data In

### Storage Element: Register File

- **Register File consists of 32 registers**:
  - Two 32-bit output busses: busA and busB
  - One 32-bit input bus: busW
- **Register is selected by**:
  - RA (number) selects the register to put on busA (data)
  - RB (number) selects the register to put on busB (data)
  - RW (number) selects the register to be written via busW (data) when Write Enable is 1
- **Clock input (CLK)**
  - The CLK input is a factor ONLY during write operation
  - During read operation, behaves as a combinational logic block:
    - RA or RB valid ⇒ busA or busB valid after “access time.”

### Storage Element: Idealized Memory

- **Memory** (idealized)
  - One input bus: Data In
  - One output bus: Data Out
- **Memory word is selected by**:
  - Address selects the word to put on Data Out
  - Write Enable = 1: address selects the memory word to be written via the Data In bus
- **Clock input (CLK)**
  - The CLK input is a factor ONLY during write operation
  - During read operation, behaves as a combinational logic block:
    - Address valid ⇒ Data Out valid after “access time.”

### Clocking Methodology

- All storage elements are clocked by the same clock edge
- Cycle Time = CLK-to-Q + Longest Delay Path + Setup + Clock Skew
- (CLK-to-Q + Shortest Delay Path - Clock Skew) > Hold Time
Step 3: Assemble DataPath meeting our requirements

- Register Transfer Requirements ⇒ Datapath Assembly
- Instruction Fetch
- Read Operands and Execute Operation

3a: Overview of the Instruction Fetch Unit

- The common RTL operations
  - Fetch the Instruction: mem[PC]
  - Update the program counter:
    - Sequential Code: PC ← PC + 4
    - Branch and Jump: PC ← "something else"

3b: Add & Subtract

- \( R[d] \leftarrow R[r] \text{ op } R[t] \)
- Example: add \( U \) rd, rs, rt
- \( Ra, Rb, \text{ and } Rw \) come from instruction’s rs, rt, and rd fields
- ALUctr and RegWr: control logic after decoding the instruction

32-bit Register-Register Timing: One complete cycle
### 3c: Logical Operations with Immediate

\[ R[rt] <- R[rs] \text{ op } \text{ZeroExt}[\text{imm16}] \]

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>26</td>
<td>21</td>
<td>16</td>
</tr>
</tbody>
</table>

### 3d: Load Operations

\[ R[rt] <- \text{Mem}[R[rs] + \text{SignExt}[\text{imm16}]] \]

Example: \text{lw} rt, rs, imm16

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>26</td>
<td>21</td>
<td>16</td>
</tr>
</tbody>
</table>

### 3e: Store Operations

\[ \text{Mem}[R[rs] + \text{SignExt}[\text{imm16}]] <- R[rt] \]

Example: \text{sw} rt, rs, imm16

### 3f: The Branch Instruction

\[ \text{beq} \ rs, rt, \text{imm16} \]

- \text{mem}[PC] — Fetch the instruction from memory
- \text{Equal} <- R[rs] == R[rt] — Calculate the branch condition
- if (Equal)
  - \text{PC} <- \text{PC} + 4 + (\text{SignExt}[\text{imm16}] \times 4)
- else
  - \text{PC} <- \text{PC} + 4
Datapath for Branch Operations

° beq rs, rt, imm16

Datapath generates condition (equal)

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>26</td>
<td>21</td>
<td>16</td>
</tr>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

9/20/99 ©UCB Fall 1999 CS152 / Kubiatowicz Lec7.37

Putting it All Together: A Single Cycle Datapath

Instruction<31:0>  Adr
Rs | Rt | Rd | Imm16

Equal
ALUOut MemWr MemToReg

9/20/99 ©UCB Fall 1999 CS152 / Kubiatowicz Lec7.38

An Abstract View of the Critical Path

° Register file and ideal memory:
  • The CLK input is a factor ONLY during write operation
  • During read operation, behave as combinational logic:
    - Address valid => Output valid after “access time.”

Critical Path (Load Operation) =
PC’s Clk-to-Q +
Instruction Memory’s Access Time +
Register File’s Access Time +
Mux to Perform a 32-bit Add +
Data Memory Access Time +
Setup Time for Register File Write +
Clock Skew

9/20/99 ©UCB Fall 1999 CS152 / Kubiatowicz Lec7.39

An Abstract View of the Implementation

9/20/99 ©UCB Fall 1999 CS152 / Kubiatowicz Lec7.40
Step 4: Given Datapath: RTL -> Control

Meaning of the Control Signals (see slide 38)

- Rs, Rt, Rd and Immed16 hardwired into datapath
- \( nPC\_sel: \ 0 \Rightarrow PC \leftarrow PC + 4; \ 1 \Rightarrow PC \leftarrow PC + 4 + \text{SignExt}(\text{Immed}16) || 00 \)

Control Signals

<table>
<thead>
<tr>
<th>Inst</th>
<th>Register Transfer</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>( R[rd] \leftarrow R[rs] + R[rt] ); PC \leftarrow PC + 4</td>
</tr>
<tr>
<td>SUB</td>
<td>( R[rd] \leftarrow R[rs] \sim R[rt] ); PC \leftarrow PC + 4</td>
</tr>
<tr>
<td>ORi</td>
<td>( R[rt] \leftarrow R[rs] + \text{zero}_\text{ext}(\text{Immed}16) ); PC \leftarrow PC + 4</td>
</tr>
<tr>
<td>LOAD</td>
<td>( R[rt] \leftarrow \text{MEM}[ R[rs] + \text{sign}_\text{ext}(\text{Immed}16)] ); PC \leftarrow PC + 4</td>
</tr>
<tr>
<td>STORE</td>
<td>( \text{MEM}[ R[rs] + \text{sign}_\text{ext}(\text{Immed}16)] \leftarrow R[rs] ); PC \leftarrow PC + 4</td>
</tr>
<tr>
<td>BEQ</td>
<td>if ( ( R[rs] \sim R[rt] ) ) then PC \leftarrow PC + \text{sign}_\text{ext}(\text{Immed}16)</td>
</tr>
</tbody>
</table>

Meaning of the Control Signals

- ExtOp: “zero”, “sign”
- ALUsrc: 0 \Rightarrow \text{regB}; 1 \Rightarrow \text{immed}
- ALUctr: “add”, “sub”, “or”
- RegDst: 0 \Rightarrow “rt”; 1 \Rightarrow “rd”
- RegWr: write dest register
- MemWr: write memory
- MemtoReg: 1 \Rightarrow \text{Mem}
- ExtOp: “zero”, “sign”
- ALUsrc = RegB, ALUctr = “add”, RegDst = rd, RegWr, nPC\_sel = “+4”
- SUB R[rd] \leftarrow R[rs] \sim R[rt]; PC \leftarrow PC + 4
- ORi R[rt] \leftarrow R[rs] + zero\_\text{ext}(\text{Immed}16); PC \leftarrow PC + 4
- LOAD R[rt] \leftarrow \text{MEM}[ R[rs] + \text{sign}\_\text{ext}(\text{Immed}16)]; PC \leftarrow PC + 4
- STORE \text{MEM}[ R[rs] + \text{sign}\_\text{ext}(\text{Immed}16)] \leftarrow R[rs]; PC \leftarrow PC + 4
- BEQ if ( R[rs] \sim R[rt] ) then PC \leftarrow PC + \text{sign}\_\text{ext}(\text{Immed}16) || 00 else PC \leftarrow PC + 4
Control Signals (Answer)

<table>
<thead>
<tr>
<th>inst</th>
<th>Register Transfer</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>( R[rd] \leftarrow R[rs] + R[rt]; ) ( PC \leftarrow PC + 4 )</td>
</tr>
<tr>
<td>SUB</td>
<td>( R[rd] \leftarrow R[rs] - R[rt]; ) ( PC \leftarrow PC + 4 )</td>
</tr>
<tr>
<td>ORi</td>
<td>( R[rt] \leftarrow R[rs] + \text{zero_ext}(\text{Imm16}); ) ( PC \leftarrow PC + 4 )</td>
</tr>
<tr>
<td>LOAD</td>
<td>( \text{MEM}[R[rs] + \text{sign_ext}(\text{Imm16})] \leftarrow R[rs]; ) ( PC \leftarrow PC + 4 )</td>
</tr>
<tr>
<td>STORE</td>
<td>( \text{MEM}[R[rs] + \text{sign_ext}(\text{Imm16})] \leftarrow R[rs]; ) ( PC \leftarrow PC + 4 )</td>
</tr>
</tbody>
</table>

Step 5: Logic for each control signal

* nPC_sel \( \leftarrow \) if (OP == BEQ) then EQUAL else 0
* ALUsrc \( \leftarrow \) if (OP == “000000”) then “regB” else “immed”
* ALUctr \( \leftarrow \) if (OP == “000000”) then funct elseif (OP == ORi) then “OR” elseif (OP == BEQ) then “sub” else “add”
* ExtOp \( \leftarrow \) _____________
* MemWr \( \leftarrow \) _____________
* MemtoReg \( \leftarrow \) _____________
* RegWr: \( \leftarrow \) _____________
* RegDst: \( \leftarrow \) _____________

Example: Load Instruction

![Example Diagram](image-url)
An Abstract View of the Implementation

 Logical vs. Physical Structure

Summary

5 steps to design a processor
- 1. Analyze instruction set => datapath requirements
- 2. Select set of datapath components & establish clock methodology
- 3. Assemble datapath meeting the requirements
- 4. Analyze implementation of each instruction to determine setting of control points that effects the register transfer.
- 5. Assemble the control logic

MIPS makes it easier
- Instructions same size
- Source registers always in same place
- Immediates same size, location
- Operations always on registers/immediates

Single cycle datapath => CPI=1, CCT => long
Next time: implementing control