32ビット RISC [ ] ( )
Contents

- Computer Component
- Performance
- Instruction Set Architecture
- ALU
- Datapath and Control
- Pipeline
- Instruction Level Parallelism
- Reference
Computer Component

• **Input**
  – Mouse, Keyboard

• **Output**
  – Display, Printer

• **Memory**
  – HDD, DRAM, SRAM, CD, DVD

• **Network**

• **Processor**
  – Datapath, Control
  – implemented using millions of transistors
  – Impossible to understand by looking at each transistor
Abstraction

- Delving into the depths reveals more information
- An abstraction omits unneeded detail, helps us cope with complexity

What are some of the details that appear in these familiar abstractions?
H/W, S/W Hierarchical Structure
The Instruction Set: a Critical Interface

Software

Hardware

Instruction set
ISA (Instruction Set Architecture)

• A very important abstraction
  – interface between hardware and low-level software
  – standardizes instructions, machine language bit patterns, etc.
  – advantage: different implementations of the same architecture
  – disadvantage: sometimes prevents using new innovations

  True or False: Binary compatibility is extraordinarily important?

• Modern instruction set architectures:
  – 80x86/Pentium/K6, PowerPC, DEC Alpha, MIPS, SPARC, HP
Execution Cycle

- **Fetch**
  - Obtain instruction from program storage

- **Decode**
  - Determine required actions and instruction size

- **Operand Fetch**
  - Locate and obtain operand data

- **Execute**
  - Compute result value or status

- **Store**
  - Deposit results in storage for later use

- **Next Instruction**
  - Determine successor instruction
ISA Classes

- **Accumulator**
  - 1 address
  - add A
  - acc <- acc+mem[A]
  - 1+ x address
  - addx A
  - acc <- acc+mem[A+x]

- **Stack**
  - 0 address
  - add
  - tos <- tos+next

- **General Purpose Register**
  - 2 address
  - add A B
  - A <- A+B
  - 3 address
  - add A B C
  - A <- A+B

- **Load/Store**
  - 3 address
  - add Ra Rb Rc
  - Ra <- Rb+Rc
  - load Ra Rb
  - Ra <- mem(Rb)
  - store Ra Rb
  - mem(Rb) <- Ra

- **Comparison**
  - Bytes per instruction? Number of instructions? Cycles per instruction?
General Purpose Registers Dominate

° 1975-1999 all machines use general purpose registers

° Advantages of registers
  • registers are faster than memory
  • registers are easier for a compiler to use
    - e.g., \((A*B) - (C*D) - (E*F)\) can do multiplies in any order vs. stack
  • registers can hold variables
    - memory traffic is reduced, so program is sped up (since registers are faster than memory)
    - code density improves (since register named with fewer bits than memory location)
Forces on Computer Architecture

Technology

Applications

Programming Languages

Operating Systems

History

\( A = \frac{F}{M} \)
Performance

• Measure, Report, and Summarize
• Make intelligent choices
• See through the marketing hype
• Key to understanding underlying organizational motivation

Why is some hardware better than others for different programs?

What factors of system performance are hardware related? (e.g., Do we need a new machine, or a new operating system?)

How does the machine's instruction set affect performance?
Computer Performance

• **Response Time (latency)**
  – How long does it take for my job to run?
  – How long does it take to execute a job?
  – How long must I wait for the database query?

• **Throughput**
  – How many jobs can the machine run at once?
  – What is the average execution rate?
  – How much work is getting done?

*If we upgrade a machine with a new processor what do we increase?*
*If we add a new machine to the lab what do we increase?*
Execution Time

- **Elapsed Time**
  - counts everything *(disk and memory accesses, I/O, etc.)*
  - a useful number, but often not good for comparison purposes

- **CPU time**
  - doesn't count I/O or time spent running other programs
  - can be broken up into system time, and user time

- **Our focus: user CPU time**
  - time spent executing the lines of code that are "in" our program
Clock Cycles

- Instead of reporting execution time in seconds, we often use cycles
  \[
  \frac{\text{seconds}}{\text{program}} = \frac{\text{cycles}}{\text{program}} \times \frac{\text{seconds}}{\text{cycle}}
  \]

- Clock “ticks” indicate when to start activities (one abstraction):

- cycle time = time between ticks = seconds per cycle
- clock rate (frequency) = cycles per second (1 Hz. = 1 cycle/sec)

A 200 Mhz. clock has a
\[
\frac{1}{200 \times 10^6} \times 10^9 = 5 \text{ nanoseconds cycle time}
\]
How to Improve Performance

\[
\frac{\text{seconds}}{\text{program}} = \frac{\text{cycles}}{\text{program}} \times \frac{\text{seconds}}{\text{cycle}}
\]

- So, to improve performance (everything else being equal) you can either
  - the number of required cycles for a program, or
  - the clock cycle time or, said another way,
  - the clock rate.
Now that we understand cycles

- A given program will require
  - some number of instructions (machine instructions)
  - some number of cycles
  - some number of seconds

- We have a vocabulary that relates these quantities:
  - cycle time (seconds per cycle)
  - clock rate (cycles per second)
  - CPI (cycles per instruction)
    - a floating point intensive application might have a higher CPI
  - MIPS (millions of instructions per second)
    - this would be higher for a program using simple instructions
Performance

• Performance is determined by execution time
• Do any of the other variables equal performance?
  – number of cycles to execute program?
  – number of instructions in program?
  – number of cycles per second?
  – average number of cycles per instruction?
  – average number of instructions per second?

• Common pitfall: thinking one of the variables is indicative of performance when it really isn’t
Aspects of CPU Performance

\[
\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}
\]

<table>
<thead>
<tr>
<th></th>
<th>instr count</th>
<th>CPI</th>
<th>clock rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compiler</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Instr. Set</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Organization</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Technology</td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>
Amdahl's Law

Execution Time After Improvement =

\[
\text{Execution Time Unaffected} + \left( \frac{\text{Execution Time Affected}}{\text{Amount of Improvement}} \right)
\]

• Example:

"Suppose a program runs in 100 seconds on a machine, with multiply responsible for 80 seconds of this time. How much do we have to improve the speed of multiplication if we want the program to run 4 times faster?"

How about making it 5 times faster?

• Principle: Make the common case fast
Instruction Set Architecture
Instructions:

- Language of the Machine
- More primitive than higher level languages
  e.g., no sophisticated control flow
- Very restrictive
  e.g., MIPS Arithmetic Instructions

- We’ll be working with the MIPS instruction set architecture
  – similar to other architectures developed since the 1980's
  – used by NEC, Nintendo, Silicon Graphics, Sony

*Design goals: maximize performance and minimize cost, reduce design time*
# RISC vs. CISC

<table>
<thead>
<tr>
<th>CISC</th>
<th>RISC</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Price/Performance Strategies</strong></td>
<td><strong>Price/Performance Strategies</strong></td>
</tr>
<tr>
<td><strong>Price</strong>: move complexity from software to hardware.</td>
<td><strong>Price</strong>: move complexity from hardware to software</td>
</tr>
<tr>
<td><strong>Performance</strong>: make tradeoffs in favor of decreased code size, at</td>
<td><strong>Performance</strong>: make tradeoffs in favor of a lower CPI, at the</td>
</tr>
<tr>
<td>the expense of a higher CPI</td>
<td>expense of increased code size.</td>
</tr>
<tr>
<td><strong>Design Decisions</strong></td>
<td><strong>Design Decisions</strong></td>
</tr>
<tr>
<td>- A large and varied instruction set that includes simple, fast</td>
<td>- Simple, single-cycle instructions that perform only basic</td>
</tr>
<tr>
<td>instructions for performing basic tasks, as well as complex,</td>
<td>functions. Assembler instructions correspond to microcode</td>
</tr>
<tr>
<td>multi-cycle instructions that correspond to statements in an</td>
<td>instructions on a CISC machine.</td>
</tr>
<tr>
<td>HLL(High Level Language)</td>
<td>- All HLL support is done in software.</td>
</tr>
<tr>
<td>- Support for HLLs is done in hardware.</td>
<td>- Simple addressing modes that allow only LOAD and STORE to</td>
</tr>
<tr>
<td>- Memory-to-memory addressing modes.</td>
<td>access memory. All operations are register-to-register.</td>
</tr>
<tr>
<td>- A microcode control unit.</td>
<td>- direct execution control unit.</td>
</tr>
<tr>
<td>- Spend fewer transistors on registers</td>
<td>- spend more transistors on multiple banks of registers.</td>
</tr>
<tr>
<td></td>
<td>- use pipelined execution to lower CPI.</td>
</tr>
</tbody>
</table>
MIPS arithmetic

• All instructions have 3 operands
• Operand order is fixed (destination first)
• Design Principle: simplicity favors regularity. Why?
• Of course this complicates some things...
  – Example:

  C code: 
  
  A = B + C + D;
  E = F - A;

  MIPS code: 
  
  add $t0, $s1, $s2
  add $s0, $t0, $s3
  sub $s4, $s5, $s0

• Operands must be registers, only 32 registers provided
• Design Principle: smaller is faster. Why?
Registers vs. Memory

- Arithmetic instructions operands must be registers, — only 32 registers provided
- Compiler associates variables with registers
- What about programs with lots of variables

![Diagram of processor components]

- Control
- Datapath
- Memory
- Input
- Output
- Processor
- I/O
Memory Organization

- Viewed as a large, single-dimension array, with an address.
- A memory address is an index into the array.
- "Byte addressing" means that the index points to a byte of memory.
- Bytes are nice, but most data items use larger "words".
- For MIPS, a word is 32 bits or 4 bytes.

<table>
<thead>
<tr>
<th>0</th>
<th>8 bits of data</th>
<th>0</th>
<th>32 bits of data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8 bits of data</td>
<td>4</td>
<td>32 bits of data</td>
</tr>
<tr>
<td>2</td>
<td>8 bits of data</td>
<td>8</td>
<td>32 bits of data</td>
</tr>
<tr>
<td>3</td>
<td>8 bits of data</td>
<td>12</td>
<td>32 bits of data</td>
</tr>
<tr>
<td>4</td>
<td>8 bits of data</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>8 bits of data</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>8 bits of data</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Registers hold 32 bits of data
MIPS Instructions

- Load and store instructions
- Example:

  
  MIPS code:  
  \[
  lw \ $t0, 32($s3) \\
  add \ $t0, \ $s2, \ $t0 \\
  sw \ $t0, 32($s3)
  \]

- Store word has destination last
- Remember arithmetic operands are registers, not memory!
MIPS Instructions

- MIPS
  - loading words but addressing bytes
  - arithmetic on registers only

- Instruction | Meaning
  - add $s1, $s2, $s3 | $s1 = $s2 + $s3
  - sub $s1, $s2, $s3 | $s1 = $s2 - $s3
  - lw $s1, 100($s2) | $s1 = Memory[$s2+100]
  - sw $s1, 100($s2) | Memory[$s2+100] = $s1
Machine Language

• Instructions, like registers and words of data, are also 32 bits long
  – Example: `add $t0, $s1, $s2`
  – registers have numbers, $t0=9$, $s1=17$, $s2=18$

• Instruction Format:

  \[
  \begin{array}{cccccc}
  \text{op} & \text{rs} & \text{rt} & \text{rd} & \text{shamt} & \text{funct} \\
  000000 & 10001 & 10010 & 01001 & 00000 & 100000 \\
  \end{array}
  \]

• *Can you guess what the field names stand for?*
Machine Language

• Consider the load-word and store-word instructions,
  – What would the regularity principle have us do?
  – New principle: Good design demands a compromise

• Introduce a new type of instruction format
  – I-type for data transfer instructions
  – other format was R-type for register

• Example: `lw $t0, 32($s2)`

```
100011 10010 01001 0000000000100000
```

op    rs    rt   16 bit number

• Where's the compromise?
Machine Language

- **Decision making instructions**
  - alter the control flow,
  - i.e., change the "next" instruction to be executed

- **MIPS conditional branch instructions:**
  
  bne $t0, $t1, Label  
  beq $t0, $t1, Label

- **Example:** if (i==j) h = i + j;

  bne $s0, $s1, Label  
  add $s3, $s0, $s1
  
  Label: ....
Constants

• Small constants are used quite frequently (50% of operands)
  e.g., \[ A = A + 5; \]
  \[ B = B + 1; \]
  \[ C = C - 18; \]
• Solutions? Why not?
  – put 'typical constants' in memory and load them.
  – create hard-wired registers (like $zero) for constants like one.

• MIPS Instructions:
  ```
  addi $29, $29, 4
  slti $8, $18, 10
  andi $29, $29, 6
  ori $29, $29, 4
  ```
• How do we make this work?
larger constants

• We'd like to be able to load a 32 bit constant into a register
• Must use two instructions, new "load upper immediate" instruction
  \[
  \text{\texttt{lui } } t0, 1010101010101010
  \]
  filled with zeros

\[
\begin{array}{c|c}
1010101010101010 & 0000000000000000 \\
\end{array}
\]

• Then must get the lower order bits right, i.e.,
  \[
  \text{\texttt{ori } } t0, t0, 1010101010101010
  \]

\[
\begin{array}{c|c}
1010101010101010 & 0000000000000000 \\
0000000000000000 & 1010101010101010 \\
\end{array}
\]

\[
\begin{array}{c|c}
1010101010101010 & 1010101010101010 \\
\end{array}
\]
Overview of MIPS

- simple instructions all 32 bits wide
- very structured, no unnecessary baggage
- only three instruction formats

<table>
<thead>
<tr>
<th>R</th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td></td>
<td>16 bit address</td>
<td></td>
</tr>
<tr>
<td>J</td>
<td>op</td>
<td></td>
<td></td>
<td></td>
<td>26 bit address</td>
<td></td>
</tr>
</tbody>
</table>

- rely on compiler to achieve performance
  — what are the compiler's goals?
- help compiler where we can
# MIPS Instructions (R type)

- **MIPS Arithmetic Operations**
  - **Instruction** | **Meaning**
  - `add $s1,$s2,$s3` | $s1 = s2 + s3$
  - `sub $s1,$s2,$s3` | $s1 = s2 - s3$
  - `slt $s1,$s2,$s3` | Compare less than; for `beq`, `bne`

- **Formats:**

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

|  |  |  |  |  |  |  |
### MIPS Instructions (I type)

- **Instruction**
  - `lw $s1,100($s2)`
  - $s1 = Memory[$s2+100]
  - `sw $s1,100($s2)`
  - Memory[$s2+100] = $s1
  - `bne $s4,$s5,L` Next instr. is at Label if $s4=$s5
  - `beq $s4,$s5,L` Next instr. is at Label if $s4=$s5
  - `addi $t1,$t1,1` $t1 = t1 + 1

- **Formats:**

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>16 bit address</th>
</tr>
</thead>
</table>

---

**MIPS Instructions (I type)**

- **Instruction**
  - `lw $s1,100($s2)` $s1 = Memory[$s2+100]
  - `sw $s1,100($s2)` Memory[$s2+100] = $s1
  - `bne $s4,$s5,L` Next instr. is at Label if $s4=$s5
  - `beq $s4,$s5,L` Next instr. is at Label if $s4=$s5
  - `addi $t1,$t1,1` $t1 = t1 + 1

- **Formats:**

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>16 bit address</th>
</tr>
</thead>
</table>
MIPS Instructions (J type)

- Offer Long addresses (26bit + 2bit) for procedure call
- Instruction | Meaning
  | - j Label | Next instr. is at Label

- Formats:

<table>
<thead>
<tr>
<th>J</th>
<th>op</th>
<th>26 bit address</th>
</tr>
</thead>
</table>

Assembly Language vs. Machine Language

- **Assembly provides convenient symbolic representation**
  - much easier than writing down numbers
  - e.g., destination first
- **Machine language is the underlying reality**
  - e.g., destination is no longer first
- **Assembly can provide 'pseudoinstructions'**
  - e.g., “move $t0, $t1” exists only in Assembly
  - would be implemented using “add $t0,$t1,$zero”
- **When considering performance you should count real instructions**
Addresses in Branches and Jumps

- **Instructions:**
  - `bne $t4,$t5,Label`  
    Next instruction is at Label if $t4 \neq t5$
  - `beq $t4,$t5,Label`  
    Next instruction is at Label if $t4 = t5$
  - `j Label`  
    Next instruction is at Label

- **Formats:**

<table>
<thead>
<tr>
<th>I</th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>16 bit address</th>
</tr>
</thead>
<tbody>
<tr>
<td>J</td>
<td>op</td>
<td></td>
<td></td>
<td>26 bit address</td>
</tr>
</tbody>
</table>

- **Addresses are not 32 bits**
  — How do we handle this with load and store instructions?
Addresses in Branches

- **Instructions:**
  - `bne $t4,$t5,Label`  Next instruction is at Label if $t4!=$t5
  - `beq $t4,$t5,Label`  Next instruction is at Label if $t4=$t5

- **Formats:**
  
<table>
<thead>
<tr>
<th>I</th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>16 bit address</th>
</tr>
</thead>
</table>

- **Could specify a register (like lw and sw) and add it to address**
  - use Instruction Address Register (PC = program counter)
  - most branches are local (principle of locality)

- **Jump instructions just use high order bits of PC**
  - address boundaries of 256 MB
Addressing Mode

- **Addressing mode**
  - **Example**
  - **Meaning**

- **Register**
  - Add R4,R3
  - R4 ← R4+R3

- **Immediate**
  - Add R4,#3
  - R4 ← R4+3

- **Displacement**
  - Add R4,100(R1)
  - R4 ← R4+Mem[100+R1]

- **Register indirect**
  - Add R4,(R1)
  - R4 ← R4+Mem[R1]

- **Indexed**
  - Add R3,(R1+R2)
  - R3 ← R3+Mem[R1+R2]

- **Direct or absolute**
  - Add R1,(1001)
  - R1 ← R1+Mem[1001]

- **Memory indirect**
  - Add R1,@(R3)
  - R1 ← R1+Mem[Mem[R3]]

- **Auto-increment**
  - Add R1,(R2)+
  - R1 ← R1+Mem[R2]; R2 ← R2+d

- **Auto-decrement**
  - Add R1,–(R2)
  - R2 ← R2–d; R1 ← R1+Mem[R2]

- **Scaled**
  - Add R1,100(R2)[R3]
  - R1 ← R1+Mem[100+R2+R3*d]
Addressing Mode

1. Immediate Addressing

\[
\begin{array}{ccc}
\text{op} & \text{rs} & \text{rt} & \text{immediate} \\
\end{array}
\]

2. Register Addressing

\[
\begin{array}{ccccccc}
\text{op} & \text{rs} & \text{rt} & \text{rd} & \ldots & \text{func} \\
\end{array}
\]

3. Base Addressing

\[
\begin{array}{cccc}
\text{op} & \text{rs} & \text{rt} & \text{address} \\
\end{array}
\]

Memory

\[
\begin{array}{ccc}
\text{Byte} & \text{Halfword} & \text{Word} \\
\end{array}
\]
Addressing Mode

4. PC Relative Addressing

```
| op | rs | rt | address |
```

PC + Word

5. Pseudodirect Addressing

```
| op | address |
```

PC + Word
Summary: Salient features of MIPS I

- **32-bit fixed format inst** (3 formats)
- **32 32-bit GPR** (R0 contains zero) and 32 FP registers (and HI LO)
  - partitioned by software convention
- **3-address, reg-reg arithmetic instr.**
- **Single address mode for load/store:** base+displacement
  - no indirection, scaled
- **16-bit immediate plus LUI**
- **Simple branch conditions**
  - compare against zero or two registers for =, ≠
  - no integer condition codes
- **Delayed branch**
  - execute instruction after a branch (or jump) even if the branch is taken
    (Compiler can fill a delayed branch with useful work about 50% of the time)
Summary: Instruction set design (MIPS)

- Use general purpose registers with a load-store architecture: **YES**
- Provide at least 16 general purpose registers plus separate floating-point registers: **31 GPR & 32 FPR**
- Support basic addressing modes: displacement (with an address offset size of 12 to 16 bits), immediate (size 8 to 16 bits), and register deferred: **YES: 16 bits for immediate, displacement (disp=0 => register deferred)**
- All addressing modes apply to all data transfer instructions: **YES**
- Use fixed instruction encoding if interested in performance and use variable instruction encoding if interested in code size: **Fixed**
- Support these data sizes and types: 8-bit, 16-bit, 32-bit integers and 32-bit and 64-bit IEEE 754 floating point numbers: **YES**
- Support these simple instructions, since they will dominate the number of instructions executed: load, store, add, subtract, move register-register, and, shift, compare equal, compare not equal, branch (with a PC-relative address at least 8-bits long), jump, call, and return: **YES, 16b**
- Aim for a minimalist instruction set: **YES**
Arithmetic Logic Unit
Arithmetic

- **Where we've been:**
  - Performance (seconds, cycles, instructions)
  - Abstractions:
    - Instruction Set Architecture
    - Assembly Language and Machine Language

- **What's up ahead:**
  - Implementing the Architecture
Numbers

- Bits are just bits (no inherent meaning) conventions define relationship between bits and numbers
- Binary numbers (base 2)
  - 0000 0001 0010 0011 0100 0101 0110 0111 1000 1001...
  - decimal: 0...2^n-1
- Of course it gets more complicated:
  - numbers are finite (overflow)
  - fractions and real numbers
  - negative numbers
  - e.g., no MIPS subi instruction; addi can add a negative number
- How do we represent negative numbers?
  - i.e., which bit patterns will represent which numbers?
MIPS

• 32 bit signed numbers:

\[
\begin{align*}
0000 & \quad 0000 \quad 0000 \quad 0000 \quad 0000 \quad 0000 \quad 0000 \quad 0000_{\text{two}} = 0_{\text{ten}} \\
0000 & \quad 0000 \quad 0000 \quad 0000 \quad 0000 \quad 0000 \quad 0000 \quad 0001_{\text{two}} = +1_{\text{ten}} \\
0000 & \quad 0000 \quad 0000 \quad 0000 \quad 0000 \quad 0000 \quad 0000 \quad 0010_{\text{two}} = +2_{\text{ten}} \\
\cdots & \\
0111 & \quad 1111 \quad 1111 \quad 1111 \quad 1111 \quad 1111 \quad 1111 \quad 1110_{\text{two}} = +2,147,483,646_{\text{ten}} \\
0111 & \quad 1111 \quad 1111 \quad 1111 \quad 1111 \quad 1111 \quad 1111 \quad 1111_{\text{two}} = +2,147,483,647_{\text{ten}} \\
1000 & \quad 0000 \quad 0000 \quad 0000 \quad 0000 \quad 0000 \quad 0000 \quad 0000_{\text{two}} = -2,147,483,648_{\text{ten}} \\
1000 & \quad 0000 \quad 0000 \quad 0000 \quad 0000 \quad 0000 \quad 0000 \quad 0001_{\text{two}} = -2,147,483,647_{\text{ten}} \\
1000 & \quad 0000 \quad 0000 \quad 0000 \quad 0000 \quad 0000 \quad 0000 \quad 0010_{\text{two}} = -2,147,483,646_{\text{ten}} \\
\cdots & \\
1111 & \quad 1111 \quad 1111 \quad 1111 \quad 1111 \quad 1111 \quad 1111 \quad 1101_{\text{two}} = -3_{\text{ten}} \\
1111 & \quad 1111 \quad 1111 \quad 1111 \quad 1111 \quad 1111 \quad 1111 \quad 1110_{\text{two}} = -2_{\text{ten}} \\
1111 & \quad 1111 \quad 1111 \quad 1111 \quad 1111 \quad 1111 \quad 1111 \quad 1111_{\text{two}} = -1_{\text{ten}}
\end{align*}
\]
Addition & Subtraction

• Just like in grade school (carry/borrow 1s)

\[
\begin{array}{c}
0111 \\
+ 110 \\
\hline
1010
\end{array}
\]

• Two's complement operations easy
  
  – subtraction using addition of negative numbers

\[
\begin{array}{c}
0111 \\
+ 0010 \\
\hline
1001
\end{array}
\]

• Overflow (result too large for finite computer word):
  
  – e.g., adding two n-bit numbers does not yield an n-bit number

\[
\begin{array}{c}
0111 \\
+ 0001 \\
\hline
1000
\end{array}
\]

\[\text{note that overflow term is somewhat misleading, it does not mean a carry overflowed}\]
Detecting Overflow

• No overflow when adding a positive and a negative number
• No overflow when signs are the same for subtraction
• Overflow occurs when the value affects the sign:
  – overflow when adding two positives yields a negative
  – or, adding two negatives gives a positive
  – or, subtract a negative from a positive and get a negative
  – or, subtract a positive from a negative and get a positive
• Consider the operations A + B, and A - B
  – Can overflow occur if B is 0?
  – Can overflow occur if A is 0?
Effects of Overflow

- An exception (interrupt) occurs
  - Control jumps to predefined address for exception
  - Interrupted address is saved for possible resumption
- Details based on software system / language
  - example: flight control vs. homework assignment
- Don't always want to detect overflow
ALU (Arithmetic Logic Unit)

• Let's build an ALU to support the `andi` and `ori` instructions
  – we'll just build a 1 bit ALU, and use 32 of them

\[
\begin{array}{cccc}
\text{operation} & \text{op} & \text{a} & \text{b} & \text{res} \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 1 \\
1 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 1 & 1 \\
1 & 1 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 \\
\end{array}
\]

• Possible Implementation (sum-of-products):

\[
res = \overline{op} \cdot a \cdot b + \overline{op} \cdot a \cdot \overline{b} + op \cdot a \cdot \overline{b} + op \cdot a \cdot b
\]
Different Implementations

- Not easy to decide the best way to build something
  - Don't want too many inputs to a single gate
  - Don't want to have to go through too many gates
  - for our purposes, ease of comprehension is important

- Let's look at a 1-bit ALU for addition:

\[
\begin{align*}
c_{\text{out}} &= a \cdot b + a \cdot c_{\text{in}} + b \cdot c_{\text{in}} \\
\text{sum} &= a \oplus b \oplus c_{\text{in}}
\end{align*}
\]

- How could we build a 1-bit ALU for add, and, and or?
- How could we build a 32-bit ALU?
Building a 32 bit ALU
Subtraction

- Two's complement approach: just negate b and add.
- How do we negate?

- A very clever solution
Tailoring the ALU to the MIPS

- **Need to support the set-on-less-than instruction (slt)**
  - remember: slt is an arithmetic instruction
  - produces a 1 if rs < rt and 0 otherwise
  - use subtraction: (a-b) < 0 implies a < b

- **Need to support test for equality (beq $t5, $t6, $t7)**
  - use subtraction: (a-b) = 0 implies a = b
Supporting \textit{slt}

1 bit ALU

\begin{align*}
&\text{Binvert} \\
&\quad \text{CarryIn} \\
&\quad \text{Operation} \\
&\quad \text{Result} \\
&\quad \text{CarryOut}
\end{align*}

\begin{align*}
&\text{Less} \\
&\quad \text{CarryIn}
\end{align*}

\begin{align*}
&a \\
&b
\end{align*}

\begin{align*}
&0 \\
&1
\end{align*}

\begin{align*}
&0 \\
&1
\end{align*}

\begin{align*}
&0 \\
&1
\end{align*}

ALU MSB

\begin{align*}
&\text{Binvert} \\
&\quad \text{CarryIn} \\
&\quad \text{Operation} \\
&\quad \text{Result} \\
&\quad \text{CarryOut}
\end{align*}

\begin{align*}
&\text{Less} \\
&\quad \text{CarryIn}
\end{align*}

\begin{align*}
&a \\
&b
\end{align*}

\begin{align*}
&0 \\
&1
\end{align*}

\begin{align*}
&0 \\
&1
\end{align*}

\begin{align*}
&0 \\
&1
\end{align*}

\begin{align*}
&0 \\
&1
\end{align*}

\begin{align*}
&0 \\
&1
\end{align*}

\begin{align*}
&0 \\
&1
\end{align*}

ALU

\begin{align*}
&\text{Binvert} \\
&\quad \text{CarryIn} \\
&\quad \text{Operation} \\
&\quad \text{Result0} \\
&\quad \text{Less} \\
&\quad \text{CarryOut}
\end{align*}

\begin{align*}
&a \\
&b
\end{align*}

\begin{align*}
&0 \\
&1
\end{align*}

\begin{align*}
&0 \\
&1
\end{align*}

\begin{align*}
&0 \\
&1
\end{align*}

\begin{align*}
&0 \\
&1
\end{align*}

\begin{align*}
&0 \\
&1
\end{align*}

\begin{align*}
&0 \\
&1
\end{align*}

\begin{align*}
&0 \\
&1
\end{align*}

\begin{align*}
&0 \\
&1
\end{align*}

\begin{align*}
&0 \\
&1
\end{align*}

\begin{align*}
&0 \\
&1
\end{align*}

\begin{align*}
&0 \\
&1
\end{align*}

\begin{align*}
&0 \\
&1
\end{align*}

\begin{align*}
&0 \\
&1
\end{align*}

\begin{align*}
&0 \\
&1
\end{align*}

\begin{align*}
&0 \\
&1
\end{align*}
Test for equality

• Notice control lines:

000 = and
001 = or
010 = add
110 = subtract
111 = slt

• Note: zero is a 1 when the result is zero!
Overflow Detection Logic

• Carry into MSB != Carry out of MSB
  – For a N-bit ALU: Overflow = CarryIn[N-1] XOR CarryOut[N-1]
Datapath and Control
The Processor: Datapath & Control

- **Simplified to contain only:**
  - memory-reference instructions: `lw, sw`
  - arithmetic-logical instructions: `add, sub, and, or, slt`
  - control flow instructions: `beq, j`

- **Generic Implementation:**
  - use the program counter (PC) to supply instruction address
  - get the instruction from memory
  - read registers
  - use the instruction to decide exactly what to do

- **All instructions use the ALU after reading the registers**
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:
  - R-type
    - op
    - rs
    - rt
    - rd
    - shamt
    - funct
  - I-type
    - op
    - rs
    - rt
    - immediate
  - J-type
    - op
    - target address

- 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
The MIPS Subset

- **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

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>26</td>
<td>21</td>
<td>16</td>
<td>11</td>
<td>6</td>
</tr>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
</tr>
</tbody>
</table>

<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>

<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>
Logical Register Transfers

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

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>R[rd] ← R[rs] + R[rt];</th>
<th>PC ← PC + 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDU</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>R[rd] ← R[rs] − R[rt];</th>
<th>PC ← PC + 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUBU</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>R[rt] ← R[rs]</th>
<th>zero_ext(Imm16);</th>
<th>PC ← PC + 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORi</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>R[rt] ← MEM[ R[rs] + sign_ext(Imm16)];</th>
<th>PC ← PC + 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>MEM[ R[rs] + sign_ext(Imm16) ] ← R[rt];</th>
<th>PC ← PC + 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>STORE</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Instruction | if ( R[rs] == R[rt] ) then PC ← PC + 4 +sign_ext(Imm16) || 00 | |
|-------------|-------------------------------------------------|----------|
| BEQ         | else PC ← PC + 4                                |          |
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**
  - A
  - B
  - CarryIn
  - Sum
  - Carry

- **MUX**
  - A
  - B
  - Select
  - Y

- **ALU**
  - A
  - B
  - OP
  - Result
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 $\implies$ 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

- 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

- \( R[rd] \leftarrow R[rs] \ op \ R[rt] \)
  - \( Ra, Rb, \) and \( Rw \) come from instruction’s \( rs, rt, \) and \( rd \) fields
  - \( ALUctr \) and \( RegWr \): control logic after decoding the instruction

```
<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
</tr>
</tbody>
</table>
```

![Diagram of ALU and registers](image.png)
Register-Register Timing: One complete cycle

Clk-to-Q

PC

Old Value

New Value

Instruction Memory Access Time

Rs, Rt, Rd,

Op, Func

Old Value

New Value

Delay through Control Logic

ALUctr

Old Value

New Value

Register File Access Time

RegWr

Old Value

New Value

ALU Delay

busA, B

Old Value

New Value

busW

Old Value

New Value

Register Write Occurs Here

32 32-bit Registers

Rw Ra Rb

busA

busB

r

Result
3c: Logical Operations with Immediate

- \( R[rt] \leftarrow R[rs] \text{ op ZeroExt}[imm16] \)
3d: Load Operations

- \( R[rt] \leftarrow \text{Mem}[R[rs] + \text{SignExt}[imm16]] \)  
  Example: \( \text{lw} \ rt, rs, \text{imm16} \)

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

![Diagram of 32-bit registers and load operations]
3e: Store Operations

  Example: sw rt, rs, imm16

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

![Diagram showing the control, data, and address paths for store operations.]
3f: The Branch Instruction

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

- **beq rs, rt, imm16**
  - mem[PC] Fetch the instruction from memory
  - Equal <- R[rs] == R[rt] Calculate the branch condition
  - if (Equal) Calculate the next instruction’s address
    - PC <- PC + 4 + (SignExt(imm16) x 4 )
  - else
    - PC <- 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>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>
```

```
beq rs, rt, imm16

Datapath generates condition (equal)
```
Putting it All Together: A Single Cycle Datapath

Instruction<31:0>

Inst Memory

Adr

Rs Rt Rd Imm16

nPC_sel

RegDst

1 0

Rs Rt

RegWr

32 32-bit Registers

Clk

PC Ext

MemWr MemtoReg

ALUct

Data In

WrEn Adr

Data Memory

imm16

ExtOp ALUSrc
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 +
ALU to Perform a 32-bit Add +
Data Memory Access Time +
Setup Time for Register File Write +
Clock Skew
Step 4: Given Datapath: RTL- > Control

Instruction<31:0>

Inst Memory
Adr

Op Fun Rt Rs Rd Imm16

Control

nPC_sel RegWr RegDst ExtOp ALUSrc ALUctr MemWr MemtoReg

Equal

DATA PATH
Meaning of the Control Signals

- Rs, Rt, Rd and Imed16 hardwired into datapath
- nPC_sel: 0 => PC ← PC + 4; 1 => PC ← PC + 4 + \text{SignExt(Im16)} || 00
Meaning of the Control Signals

- **ExtOp:** “zero”, “sign”
- **ALUsrc:** 0 => regB; 1 => immed
- **ALUctr:** “add”, “sub”, “or”
- **MemWr:** write memory
- **MemtoReg:** 1 => Mem
- **RegDst:** 0 => “rt”; 1 => “rd”
- **RegWr:** write dest register
Control Signals

inst Register Transfer

ADD R[rd] <- R[rs] + R[rt]; PC <- PC + 4
ALUsrc = RegB, ALUctr = “add”, RegDst = rd, RegWr, nPC_sel = “+4”

SUB R[rd] <- R[rs] – R[rt]; PC <- PC + 4
ALUsrc = ___, Extop = __, ALUctr = ___, RegDst = ___, RegWr(?), MemtoReg(?), MemWr(?), nPC_sel = __

ORi R[rt] <- R[rs] + zero_ext(Imm16); PC <- PC + 4
ALUsrc = ___, Extop = __, ALUctr = ___, RegDst = ___, RegWr(?), MemtoReg(?), MemWr(?), nPC_sel = __

LOAD R[rt] <- MEM[ R[rs] + sign_ext(Imm16)]; PC <- PC + 4
ALUsrc = ___, Extop = __, ALUctr = ___, RegDst = ___, RegWr(?), MemtoReg(?), MemWr(?), nPC_sel = __

STORE MEM[ R[rs] + sign_ext(Imm16)] <- R[rs]; PC <- PC + 4
ALUsrc = ___, Extop = __, ALUctr = ___, RegDst = ___, RegWr(?), MemtoReg(?), MemWr(?), nPC_sel = __

BEQ if ( R[rs] == R[rt] ) then PC <- PC + sign_ext(Imm16)] || 00 else PC <- PC + 4
ALUsrc = ___, Extop = __, ALUctr = ___, RegDst = ___, RegWr(?), MemtoReg(?), MemWr(?), nPC_sel = __
Control Signals

**inst Register Transfer**

**ADD**  \(R[rd] \leftarrow R[rs] + R[rt]; \ \text{PC} \leftarrow \text{PC} + 4\)

\(\text{ALUsrc} = \text{RegB}, \ \text{ALUctr} = \text{“add”}, \ \text{RegDst} = \text{rd}, \ \text{RegWr}, \ \text{nPC\_sel} = \text{“+4”}\)

**SUB**  \(R[rd] \leftarrow R[rs] - R[rt]; \ \text{PC} \leftarrow \text{PC} + 4\)

\(\text{ALUsrc} = \text{RegB}, \ \text{ALUctr} = \text{“sub”}, \ \text{RegDst} = \text{rd}, \ \text{RegWr}, \ \text{nPC\_sel} = \text{“+4”}\)

**ORi** \(R[rt] \leftarrow R[rs] + \text{zero\_ext(Imm16)}; \ \ \ \text{PC} \leftarrow \text{PC} + 4\)

\(\text{ALUsrc} = \text{Im}, \ \text{Extop} = \text{“Z”}, \ \text{ALUctr} = \text{“or”}, \ \text{RegDst} = \text{rt}, \ \text{RegWr}, \ \text{nPC\_sel} = \text{“+4”}\)

**LOAD**  \(R[rt] \leftarrow \text{MEM}[ R[rs] + \text{sign\_ext(Imm16)}]; \ \ \ \text{PC} \leftarrow \text{PC} + 4\)

\(\text{ALUsrc} = \text{Im}, \ \text{Extop} = \text{“Sn”}, \ \text{ALUctr} = \text{“add”}, \\)
\(\text{MemtoReg, RegDst} = \text{rt}, \ \text{RegWr}, \ \text{nPC\_sel} = \text{“+4”}\)

**STORE**  \(\text{MEM}[ R[rs] + \text{sign\_ext(Imm16)}] \leftarrow R[rs]; \ \ \ \text{PC} \leftarrow \text{PC} + 4\)

\(\text{ALUsrc} = \text{Im}, \ \text{Extop} = \text{“Sn”}, \ \text{ALUctr} = \text{“add”}, \ \text{MemWr}, \ \text{nPC\_sel} = \text{“+4”}\)

**BEQ**  \(\text{if} \ (R[rs] \ == \ R[rt] ) \ \text{then} \ \text{PC} \leftarrow \text{PC} + \text{sign\_ext(Imm16)} \ || \ 00 \ \text{else} \ \text{PC} \leftarrow \text{PC} + 4\)

\(\text{nPC\_sel} = \text{EQUAL}, \ \text{ALUctr} = \text{“sub”}\)
Step 5: Logic for each control signal

- \( nPC\_sel \) <= if (OP == BEQ) then EQUAL else 0
- \( ALU\_src \) <= if (OP == “000000”) then “regB” else “immed”
- \( ALU\_ctr \) <= if (OP == “000000”) then funct
  elseif (OP == ORi) then “OR”
  elseif (OP == BEQ) then “sub”
  else “add”
- \( ExtOp \) <= _____________
- \( Mem\_Wr \) <= _____________
- \( Mem\_to\_Reg \) <= _____________
- \( Reg\_Wr: \) <= _____________
- \( Reg\_Dst: \) <= _____________
Step 5: Logic for each control signal

- \( nPC\text{\_sel} \) <= if (\( OP = \text{BEQ} \)) then \( \text{EQUAL} \) else 0
- \( ALU\text{src} \) <= if (\( OP = \text{“000000”} \)) then \( \text{“regB”} \) else \( \text{“immed”} \)
- \( ALU\text{ctr} \) <= if (\( OP = \text{“000000”} \)) then \( \text{funct} \)
  elseif (\( OP = \text{ORi} \)) then \( \text{“OR”} \)
  elseif (\( OP = \text{BEQ} \)) then \( \text{“sub”} \)
  else \( \text{“add”} \)
- \( \text{ExtOp} \) <= if (\( OP = \text{ORi} \)) then \( \text{“zero”} \) else \( \text{“sign”} \)
- \( \text{MemWr} \) <= (\( OP = \text{Store} \))
- \( \text{MemtoReg} \) <= (\( OP = \text{Load} \))
- \( \text{RegWr} : \) <= if ((\( OP = \text{Store} \)) || (\( OP = \text{BEQ} \))) then 0 else 1
- \( \text{RegDst} : \) <= if ((\( OP = \text{Load} \)) || (\( OP = \text{ORi} \))) then 0 else 1
An Abstract View of the Implementation

Datapath

Control

Ideal Instruction Memory

Instruction Address

Next Address

PC

Clk

32 32-bit Registers

Rw Ra Rb

A

32

B

32

Data Memory

Ideal

Address

Data

Out

A

B

Conditions

Control Signals

Instruction

Rd Rs Rt

5 5 5

Instruction

32

Ideal Instruction Memory

32

32-bit Registers

ALU

Data

In

Data

Out

Clk
A Real MIPS Datapath
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**
Recap: A Single Cycle Datapath

- We have everything except control signals (underline)
  - Today's lecture will show you how to generate the control signals
The Big Picture: Where are We Now?

- The Five Classic Components of a Computer

- Today’s Topic: Designing the Control for the Single Cycle Datapath
RTL: The \textbf{Add} Instruction

\begin{center}
\begin{tabular}{|c|c|c|c|c|c|c|}
\hline
31 & 26 & 21 & 16 & 11 & 6 & 0 \\
\hline
\textbf{op} & \textbf{rs} & \textbf{rt} & \textbf{rd} & \textbf{shamt} & \textbf{funct} \\
6 bits & 5 bits & 5 bits & 5 bits & 5 bits & 6 bits \\
\hline
\end{tabular}
\end{center}

- \textbf{add} \texttt{rd}, \texttt{rs}, \texttt{rt}
  - \texttt{mem[PC]} \quad \text{Fetch the instruction from memory}
  - \texttt{R[rd] <- R[rs] + R[rt]} \quad \text{The actual operation}
  - \texttt{PC <- PC + 4} \quad \text{Calculate the next instruction’s address}
Instruction Fetch Unit at the Beginning of Add

- Fetch the instruction from Instruction memory: Instruction <- mem[PC]
  - This is the same for all instructions
The Single Cycle Datapath during Or Immediate

- \( R[rt] \leftarrow R[rs] \) or ZeroExt[Imm16]

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

- \( R[rt] \leftarrow R[rs] \) or ZeroExt[Imm16]

The Single Cycle Datapath during Or Immediate

- \( R[rt] \leftarrow R[rs] \) or ZeroExt[Imm16]
The Single Cycle Datapath during Or Immediate

- \( R[rt] \leftarrow R[rs] \) or ZeroExt[Imm16]
The Single Cycle Datapath during Load

- \( R[rt] \leftarrow \text{Data Memory} \{ R[rs] + \text{SignExt}[\text{imm16}] \} \)

![Diagram of the single cycle datapath during load](image-url)
The Single Cycle Datapath during Store

- Data Memory \( \{R[rs] + \text{SignExt}[\text{imm16}]\} \leftarrow R[rt] \)
The Single Cycle Datapath during Store

- Data Memory \{R[rs] + SignExt[imm16]\} <- R[rt]
The Single Cycle Datapath during Branch

- if \( (R[rs] - R[rt] == 0) \) then Zero <- 1 ; else Zero <- 0
Instruction Fetch Unit at the End of Branch

- if (Zero == 1) then \( PC = PC + 4 + \text{SignExt}[\text{imm16}] \times 4 \); else \( PC = PC + 4 \)
A Summary of Control Signals

inst  Register Transfer
ADD   R[rd] ← R[rs] + R[rt]; PC ← PC + 4
       ALUsrc = RegB, ALUctr = "add", RegDst = rd, RegWr, nPC_sel = "+4"
SUB   R[rd] ← R[rs] – R[rt]; PC ← PC + 4
       ALUsrc = RegB, ALUctr = "sub", RegDst = rd, RegWr, nPC_sel = "+4"
ORi  R[rt] ← R[rs] + zero_ext(Imm16); PC ← PC + 4
       ALUsrc = Im, Extop = "Z", ALUctr = "or", RegDst = rt, RegWr, nPC_sel = "+4"
LOAD  R[rt] ← MEM[ R[rs] + sign_ext(Imm16)]; PC ← PC + 4
       ALUsrc = Im, Extop = "Sn", ALUctr = "add",
               MemtoReg, RegDst = rt, RegWr, nPC_sel = "+4"
STORE MEM[ R[rs] + sign_ext(Imm16)] ← R[rs]; PC ← PC + 4
       ALUsrc = Im, Extop = "Sn", ALUctr = "add", MemWr, nPC_sel = "+4"
BEQ   if ( R[rs] == R[rt] ) then PC ← PC + sign_ext(Imm16)] || 00 else PC ← PC + 4
       nPC_sel = "Br", ALUctr = "sub"
### A Summary of the Control Signals

<table>
<thead>
<tr>
<th>See Appendix A</th>
<th>func</th>
<th>op</th>
<th>00 0000</th>
<th>00 0010</th>
<th>We Don’t Care : - )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>add</td>
<td>sub</td>
<td>ori</td>
<td>lw</td>
<td>sw</td>
</tr>
<tr>
<td>RegDst</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>ALUSrc</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>MemtoReg</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>x</td>
</tr>
<tr>
<td>RegWrite</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>MemWrite</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>nPCsel</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Jump</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ExtOp</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ALUctr&lt;2:0&gt;</td>
<td>Add</td>
<td>Subtract</td>
<td>Or</td>
<td>Add</td>
<td>Add</td>
</tr>
</tbody>
</table>

### Instruction Formats

- **R-type**
  - op, rs, rt, rd, shamt, funct
  - Operands: add, sub

- **I-type**
  - op, rs, rt
  - Immediate Value: ori, lw, sw, beq

- **J-type**
  - op
  - Target Address: jump
# The Concept of Local Decoding

<table>
<thead>
<tr>
<th>op</th>
<th>00 0000</th>
<th>00 1101</th>
<th>10 0011</th>
<th>10 1011</th>
<th>00 0100</th>
<th>00 0010</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R-type</td>
<td>ori</td>
<td>lw</td>
<td>sw</td>
<td>beq</td>
<td>jump</td>
</tr>
<tr>
<td>RegDst</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>ALUSrc</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>MemtoReg</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>RegWrite</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MemWrite</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Branch</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Jump</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ExtOp</td>
<td>x</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>ALUop&lt;N:0&gt;</td>
<td>“R-type”</td>
<td>Or</td>
<td>Add</td>
<td>Add</td>
<td>Subtract</td>
<td>xxx</td>
</tr>
</tbody>
</table>

The diagram illustrates the flow of control and data through the processor. The `op` field is processed by the Main Control unit, which then selects the appropriate ALU control (Local) based on the `func` and `ALUop` fields. The ALU control unit generates the necessary ALU operation based on the `ALUop` and `func` values.
The Encoding of ALUop

• In this exercise, ALUop has to be 2 bits wide to represent:
  – (1) “R-type” instructions
  – “I-type” instructions that require the ALU to perform:
    • (2) Or, (3) Add, and (4) Subtract

• To implement the full MIPS ISA, ALUop has to be 3 bits to represent:
  – (1) “R-type” instructions
  – “I-type” instructions that require the ALU to perform:
    • (2) Or, (3) Add, (4) Subtract, and (5) And (Example: andi)

<table>
<thead>
<tr>
<th>ALUop (Symbolic)</th>
<th>R-type</th>
<th>ori</th>
<th>lw</th>
<th>sw</th>
<th>beq</th>
<th>jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>“R-type”</td>
<td>Or</td>
<td>Add</td>
<td>Add</td>
<td>Subtract</td>
<td>xxx</td>
<td></td>
</tr>
<tr>
<td>ALUop&lt;2:0&gt;</td>
<td>1 00</td>
<td>0 10</td>
<td>0 00</td>
<td>0 00</td>
<td>0 01</td>
<td>xxx</td>
</tr>
</tbody>
</table>
The Decoding of the “func” Field

\[
\begin{array}{|c|c|c|c|c|c|c|c|}
\hline
\text{func<5:0>} & \text{Instruction Operation} \\
\hline
10 0000 & add \\
10 0010 & subtract \\
10 0100 & and \\
10 0101 & or \\
10 1010 & set-on-less-than \\
\hline
\end{array}
\]

P. 286 text:

\[
\begin{array}{|c|c|c|c|c|c|c|}
\hline
\text{ALUctr<2:0>} & \text{ALU Operation} \\
\hline
000 & And \\
001 & Or \\
010 & Add \\
110 & Subtract \\
111 & Set-on-less-than \\
\hline
\end{array}
\]
# The Truth Table for ALUctr

<table>
<thead>
<tr>
<th>ALUop (Symbolic)</th>
<th>R-type</th>
<th>ori</th>
<th>lw</th>
<th>sw</th>
<th>beq</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>“R-type”</td>
<td>Or</td>
<td>Add</td>
<td>Add</td>
<td>Subtract</td>
</tr>
<tr>
<td>ALUop&lt;2:0&gt;</td>
<td>1 0 0</td>
<td>0 1 0</td>
<td>0 0 0</td>
<td>0 0 0</td>
<td>0 0 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>add</td>
</tr>
<tr>
<td>0010</td>
<td>subtract</td>
</tr>
<tr>
<td>0100</td>
<td>and</td>
</tr>
<tr>
<td>0101</td>
<td>or</td>
</tr>
<tr>
<td>1010</td>
<td>set-on-less-than</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ALUop</th>
<th>ALUop&lt;2:0&gt;</th>
<th>func&lt;3:0&gt;</th>
<th>Instruction Op.</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit&lt;2&gt; bit&lt;1&gt; bit&lt;0&gt;</td>
<td>bit&lt;3&gt; bit&lt;2&gt; bit&lt;1&gt; bit&lt;0&gt;</td>
<td>funct&lt;3:0&gt;</td>
<td>Instruction Op.</td>
</tr>
<tr>
<td>-------</td>
<td>------------</td>
<td>-----------</td>
<td>------------------</td>
</tr>
<tr>
<td>0 0 0</td>
<td>x x x x x</td>
<td>0000</td>
<td>add</td>
</tr>
<tr>
<td>0 x 1</td>
<td>x x x x x</td>
<td>0010</td>
<td>subtract</td>
</tr>
<tr>
<td>0 1 x</td>
<td>x x x x x</td>
<td>0100</td>
<td>and</td>
</tr>
<tr>
<td>1 x x</td>
<td>0 0 0 0 0</td>
<td>0101</td>
<td>or</td>
</tr>
<tr>
<td>1 x x</td>
<td>0 0 1 0 0</td>
<td>1010</td>
<td>set-on-less-than</td>
</tr>
<tr>
<td>1 x x</td>
<td>0 1 0 0 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 x x</td>
<td>0 1 0 1 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 x x</td>
<td>1 0 1 0 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 x x</td>
<td>1 0 1 0 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ALU Operation</th>
<th>ALUctr&lt;2:0&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit&lt;2&gt; bit&lt;1&gt; bit&lt;0&gt;</td>
<td>bit&lt;2&gt; bit&lt;1&gt; bit&lt;0&gt;</td>
</tr>
<tr>
<td>Add</td>
<td>0 1 0</td>
</tr>
<tr>
<td>Subtract</td>
<td>1 1 0</td>
</tr>
<tr>
<td>Or</td>
<td>0 0 1</td>
</tr>
<tr>
<td>Add</td>
<td>0 1 0</td>
</tr>
<tr>
<td>Subtract</td>
<td>1 1 0</td>
</tr>
<tr>
<td>And</td>
<td>0 0 0</td>
</tr>
<tr>
<td>Or</td>
<td>0 0 1</td>
</tr>
<tr>
<td>Set on &lt;</td>
<td>1 1 1</td>
</tr>
</tbody>
</table>
The Logic Equation for ALUctr<2>

\[
\text{ALUctr}<2> = \overline{\text{ALUop}<2>} \text{ & ALUop}<0> + \\
\text{ALUop}<2> \text{ & } \overline{\text{func}<2>} \text{ & \ func<1> \ & \ } \overline{\text{func}<0>}
\]

This makes func<3> a don’t care
The Logic Equation for ALUctr<1>

<table>
<thead>
<tr>
<th>ALUop bit&lt;2&gt;</th>
<th>ALUop bit&lt;1&gt;</th>
<th>ALUop bit&lt;0&gt;</th>
<th>func bit&lt;3&gt;</th>
<th>func bit&lt;2&gt;</th>
<th>func bit&lt;1&gt;</th>
<th>func bit&lt;0&gt;</th>
<th>ALUctr&lt;1&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>x</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>x</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

\[
\text{ALUctr<1>} = \neg \text{ALUop<2>} \& \neg \text{ALUop<0>} + \\
\text{ALUop<2>} \& \neg \text{func<2>} \& \neg \text{func<0>}
\]
The Logic Equation for ALUctr<0>

<table>
<thead>
<tr>
<th>ALUop</th>
<th>func</th>
<th>ALUctr&lt;0&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit&lt;2&gt; bit&lt;1&gt; bit&lt;0&gt;</td>
<td>bit&lt;3&gt; bit&lt;2&gt; bit&lt;1&gt; bit&lt;0&gt;</td>
<td></td>
</tr>
<tr>
<td>0 1 x</td>
<td>x x x x</td>
<td>1</td>
</tr>
<tr>
<td>1 x x</td>
<td>0 1 0 1</td>
<td>1</td>
</tr>
<tr>
<td>1 x x</td>
<td>1 0 1 0</td>
<td>1</td>
</tr>
</tbody>
</table>

- \( \text{ALUctr}<0> = \neg \text{ALUop}<2> \land \text{ALUop}<1> \)
  + \( \text{ALUop}<2> \land \neg \text{func}<3> \land \text{func}<2> \land \neg \text{func}<1> \land \text{func}<0> \)
  + \( \text{ALUop}<2> \land \text{func}<3> \land \neg \text{func}<2> \land \text{func}<1> \land \neg \text{func}<0> \)
The ALU Control Block

- \( \text{ALUctr} <2> = !\text{ALUop} <2> \& \text{ALUop} <0> + \)
  \( \text{ALUop} <2> \& !\text{func} <2> \& \text{func} <1> \& !\text{func} <0> \)
- \( \text{ALUctr} <1> = !\text{ALUop} <2> \& \text{ALUop} <0> + \)
  \( \text{ALUop} <2> \& !\text{func} <2> \& !\text{func} <0> \)
- \( \text{ALUctr} <0> = !\text{ALUop} <2> \& \text{ALUop} <1> \)
  \( + \text{ALUop} <2> \& \text{func} <3> \& \text{func} <2> \& !\text{func} <1> \& \text{func} <0> \)
  \( + \text{ALUop} <2> \& \text{func} <3> \& !\text{func} <2> \& \text{func} <1> \& !\text{func} <0> \)
Step 5: Logic for each control signal

- \( nPC\text{\_sel} \) \( \leftarrow \) if \( (OP == BEQ) \) then “Br” else “+4”
- \( ALU\text{src} \) \( \leftarrow \) if \( (OP == \text{“Rtype”}) \) then “regB” else “immed”
- \( ALU\text{ctr} \) \( \leftarrow \) if \( (OP == \text{“Rtype”}) \) then \( \text{funct} \) elseif \( (OP == ORi) \) then “OR” elseif \( (OP == BEQ) \) then “sub” else “add”
- \( ExtOp \) \( \leftarrow \) if \( (OP == ORi) \) then “zero” else “sign”
- \( Mem\text{Wr} \) \( \leftarrow \) (\( OP == \text{Store} \))
- \( Mem\text{toReg} \) \( \leftarrow \) (\( OP == \text{Load} \))
- \( Reg\text{Wr:} \) \( \leftarrow \) if \( ((OP == \text{Store} || (OP == BEQ)) \) then 0 else 1
- \( Reg\text{Dst:} \) \( \leftarrow \) if \( ((OP == \text{Load} || (OP == ORi)) \) then 0 else 1
The “Truth Table” for the Main Control

![Diagram of the Main Control Logic]

<table>
<thead>
<tr>
<th>op</th>
<th>00 0000</th>
<th>00 1101</th>
<th>10 0011</th>
<th>10 1011</th>
<th>00 0100</th>
<th>00 0010</th>
</tr>
</thead>
<tbody>
<tr>
<td>RegDst</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>ALUSrc</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>MemtoReg</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>RegWrite</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MemWrite</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>nPC_sel</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Jump</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ExtOp</td>
<td>x</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>ALUop (Symbolic)</td>
<td>“R-type”</td>
<td>Or</td>
<td>Add</td>
<td>Add</td>
<td>Subtract</td>
<td>xxx</td>
</tr>
<tr>
<td>ALUop &lt;2&gt;</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>ALUop &lt;1&gt;</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>ALUop &lt;0&gt;</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>x</td>
</tr>
</tbody>
</table>
The “Truth Table” for RegWrite

<table>
<thead>
<tr>
<th>op</th>
<th>R-type</th>
<th>ori</th>
<th>lw</th>
<th>sw</th>
<th>beq</th>
<th>jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 0000</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00 1101</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10 0011</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10 1011</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00 0100</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00 0010</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- RegWrite = R-type + ori + lw
  - = !op<5> & !op<4> & !op<3> & !op<2> & !op<1> & !op<0> (R-type)
  - + !op<5> & !op<4> & op<3> & op<2> & !op<1> & op<0>(ori)
  - + op<5> & !op<4> & !op<3> & !op<2> & op<1> & op<0>(lw)
PLA Implementation of the Main Control
Putting it All Together: A Single Cycle Processor

Main Control

ALUop  RegDst  ALUSrc

Instr<31:6>

3

func

Instr<5:0> 6

ALU Control

ALUctr

Instruction<31:0>

21:25 16:20 11:15 0:15

Rt  Rs  Rd  Imm16

Instruction Fetch Unit

nPC sel

Zero  MemWr  MemtoReg

Main Control

RegDst

Rd  Rt

1  0

Mux

RegWr

5  5  5

Rs  Rt

Mux

busW

32

Clk

Rw  Ra  Rb

32 32-bit Registers

busA

r

32

busB

32

Extender

imm16  Instr<15:0> 16

Instr<5:0>

ExtOp

ALUop  ALUctr

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctrl

ALUSrc

ALUop

ALUctr
Single Cycle Implementation
Worst Case Timing (Load)
Drawback of this Single Cycle Processor

- **Long cycle time:**
  - Cycle time must be long enough for the load instruction:
    - PC's Clock - to- Q +
    - Instruction Memory Access Time +
    - Register File Access Time +
    - ALU Delay (address calculation) +
    - Data Memory Access Time +
    - Register File Setup Time +
    - Clock Skew

- **Cycle time for load is much longer than needed for all other instructions**
Summary

- Single cycle datapath => CPI=1, CCT => long
- 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
- Control is the hard part
- MIPS makes control easier
  - Instructions same size
  - Source registers always in same place
  - Immediates same size, location
  - Operations always on registers/immediates
Multicycle Approach

• We will be reusing functional units
  – ALU used to compute address and to increment PC
  – Memory used for instruction and data

• Our control signals will not be determined soley by instruction
  – e.g., what should the ALU do for a subtract? instruction?

• We will use a finite state machine for control
Multicycle Approach

- Break up the instructions into steps, each step takes a cycle
  - balance the amount of work to be done
  - restrict each cycle to use only one major functional unit
- At the end of a cycle
  - store values for use in later cycles (easiest thing to do)
  - introduce additional “internal” registers
Five Execution Steps

- Instruction Fetch
- Instruction Decode and Register Fetch
- Execution, Memory Address Computation, or Branch Completion
- Memory Access or R-type instruction completion
- Write-back step

*INSTRUCTIONS TAKE FROM 3 - 5 CYCLES!*
Step 1: Instruction Fetch

• Use PC to get instruction and put it in the Instruction Register.
• Increment the PC by 4 and put the result back in the PC.
• Can be described succinctly using RTL "Register-Transfer Language"

\[
\begin{align*}
IR &= \text{Memory}[PC]; \\
PC &= PC + 4;
\end{align*}
\]

*Can we figure out the values of the control signals?*

*What is the advantage of updating the PC now?*
Step 2: Instruction Decode and Register Fetch

- Read registers rs and rt in case we need them
- Compute the branch address in case the instruction is a branch
- RTL:

  \[
  A = \text{Reg[IR[25-21]]}; \\
  B = \text{Reg[IR[20-16]]}; \\
  \text{ALUOut} = \text{PC} + (\text{sign-extend(IR[15-0])} \ll 2);
  \]

- We aren't setting any control lines based on the instruction type
  (we are busy "decoding" it in our control logic)
Step 3 (instruction dependent)

- ALU is performing one of three functions, based on instruction type

- Memory Reference:

  \[ \text{ALUOut} = A + \text{sign-extend}(\text{IR}[15-0]) \; ; \]

- R-type:

  \[ \text{ALUOut} = A \text{ op } B \; ; \]

- Branch:

  \[ \text{if } (A==B) \; \text{PC} = \text{ALUOut} \; ; \]
Step 4 (R-type or memory-access)

- Loads and stores access memory
  
  \[
  \text{MDR} = \text{Memory[ALUOut]};
  \]
  
  or
  
  \[
  \text{Memory[ALUOut]} = B;
  \]

- R-type instructions finish
  
  \[
  \text{Reg[IR[15-11]]} = \text{ALUOut};
  \]

*The write actually takes place at the end of the cycle on the edge*
Step 5 Write-back step

- \( \text{Reg[IR[20-16]]} = \text{MDR}; \)

What about all the other instructions?

<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>Instruction fetch</td>
<td></td>
<td>IR = Memory[PC]</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/jump completion</td>
<td>ALUOut = A op B</td>
<td>ALUOut = A + sign-extend (IR[15-0])</td>
<td>if (A == B) then PC = ALUOut</td>
<td>PC = PC [31-28] II (IR[25-0] &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 Store: Memory [ALUOut] = B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory read completion</td>
<td></td>
<td>Load: Reg[IR[20-16]] = MDR</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Multicycle Implement
Multicycle Implement
Multicycle Implement
Implementing the Control

- Value of control signals is dependent upon:
  - what instruction is being executed
  - which step is being performed

- Use the information we’ve accumulated to specify a finite state machine
  - specify the finite state machine graphically, or
  - use microprogramming

- Implementation can be derived from specification
FSM (Start)

Instruction fetch
0
- MemRead
- ALUSrcA = 0
- lorD = 0
- IRWrite
- ALUSrcB = 01
- ALUOp = 00
- PCWrite
- PCSource = 00

Instruction decode/ Register fetch
1
- ALUSrcA = 0
- ALUSrcB = 11
- ALUOp = 00

Start

Memory reference FSM
(Op = 'LW') or (Op = 'SW')

R-type FSM
(Op = R-type)

Branch FSM
(Op = 'BEQ')

Jump FSM
(Op = 'JMP')
FSM(LW, SW)
FSM (R-type)

From state 1
(Op = R-type)

Execution

ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10

R-type completion

RegDst = 1
RegWrite
MemtoReg = 0

To state 0
FSM(BEQ, J)
Exception Control

• Exceptions
  – Undefined Instruction
    • When no next state is defined from state 1 for the op value
    • Solution: defining the next-state value for all op values as state 10
  – Arithmetic Overflow
    • Signal called Overflow
    • Solution: Specify another additional possible next state for state 7
FSM with the additions to handle exception detection
Finite State Machine for Control

- Implementation:

```
- Instruction register
- Opcode field
- Control
- Outputs
- Inputs
- State register
```

Inputs:
- Op5
- Op4
- Op3
- Op2
- Op1
- Op0

Outputs:
- PCWrite
- PCWriteCond
- IorD
- MemRead
- MemWrite
- IRWrite
- MemtoReg
- PCSource
- ALUOp
- ALUSrcB
- ALUSrcA
- RegWrite
- RegDst
- NS3
- NS2
- NS1
- NS0

Instruction register opcode field

State register