Lecture 7: Performance Measurement & MIPS
Single-Cycle Implementation

September 22, 2009
Which has higher performance?

What is performance?
- Time to completion (latency)? – Concorde 2.2x
- Throughput? – 747 1.6x

We’re concerned with performance, but there are other, sometime more important, metrics:
- Cost
- Power
- Footprint
- Weight, ,,,
Latency of What?

- DC-Paris trip:
  - Drive to airport
  - Park
  - Take shuttle
  - Check in
  - Security
  - Wait at gate
  - Board
  - Wait on plane
  - Wait on runway
  - Fly
  - Land
  - ...

- Run application
  - Request resources
  - Get scheduled
  - Run
  - Request resources
  - ...

Total CPU time
Performance Is...

- Performance is measured in terms of things-per-second
  - Bigger is better

- CPU Latency = Execution Time (ET)

- Performance(x) = \frac{1}{\text{ExecutionTime}(x)}

- X is n times faster than y means:

  \[ n = \frac{\text{Performance}(x)}{\text{Performance}(y)} = \frac{\text{ExecutionTime}(y)}{\text{ExecutionTime}(x)} \]
Review: Machine Clock Rate

- Clock rate (MHz, GHz) is inverse of clock cycle time

\[ CC = \frac{1}{CR} \]

- 10 nsec clock cycle $\Rightarrow$ 100 MHz clock rate
- 5 nsec clock cycle $\Rightarrow$ 200 MHz clock rate
- 2 nsec clock cycle $\Rightarrow$ 500 MHz clock rate
- 1 nsec clock cycle $\Rightarrow$ 1 GHz clock rate
- 500 psec clock cycle $\Rightarrow$ 2 GHz clock rate
- 250 psec clock cycle $\Rightarrow$ 4 GHz clock rate
- 200 psec clock cycle $\Rightarrow$ 5 GHz clock rate
CPU Performance Factors

- \( ET(x) = \#CC(x) \times CC \)

- \( ET(x) = \frac{\#CC(x)}{CR} \)

- \( \#CC(x) = \#Instructions(x) \times CPI \) (Cycles per instruction)

\[
ET = \#I \times CPI \times CC
\]

We can improve ET by:
- Reducing CC (increasing CR) (Technology)
- Reducing \( \#CC \):
  - Fewer instructions (Compiler)
  - Fewer cycles per instruction (Architecture)
Which is Faster?

- Two implementations of the same instruction:

<table>
<thead>
<tr>
<th>Machine</th>
<th>CC</th>
<th>CPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1 nsec</td>
<td>2.0</td>
</tr>
<tr>
<td>B</td>
<td>1.25 nsec</td>
<td>1.2</td>
</tr>
</tbody>
</table>

- ET(A) = 1 x 2.0 x 10^{-9} = 2 \times 10^{-9} \text{ sec}
- ET(B) = 1 x 1.2 x 1.25 \times 10^{-9} = 1.5 \times 10^{-9} \text{ sec}

\[
\frac{\text{Performance}(A)}{\text{Performance}(B)} = \frac{\text{ET}(B)}{\text{ET}(A)} = \frac{1.5 \times 10^{-9}}{2.0 \times 10^{-9}} = 0.75
\]
Groups of Instructions

- Group instructions by CPI
- Consider two assembly language implementations of the same HLL code segment

<table>
<thead>
<tr>
<th>Group</th>
<th>CPI</th>
<th>#I</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>3</td>
<td>2</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Group</th>
<th>CPI</th>
<th>#I</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>3</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

Fewer instructions may not mean higher performance
To analyze program-level performance we compute the effective CPI:

\[
\text{Effective CPI} = \sum_{i=1}^{n} (\text{CPI}_i \times \text{IC}_i)
\]

<table>
<thead>
<tr>
<th>Group</th>
<th>CPI</th>
<th>Rel. Freq.</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>4</td>
<td>0.45</td>
<td>1.80</td>
</tr>
<tr>
<td>Load/Store</td>
<td>6</td>
<td>0.35</td>
<td>2.10</td>
</tr>
<tr>
<td>Test</td>
<td>3</td>
<td>0.05</td>
<td>0.15</td>
</tr>
<tr>
<td>Branch</td>
<td>3</td>
<td>0.15</td>
<td>0.45</td>
</tr>
<tr>
<td>Sum</td>
<td></td>
<td>1.00</td>
<td>4.50</td>
</tr>
</tbody>
</table>

Note that to be meaningful, the sum of the relative frequencies must be 1.0
Example 1

- **Program parameters:**
  - #I: $7.5 \times 10^9$
  - Clock rate: 600MHz
  - CPI data:

<table>
<thead>
<tr>
<th>Group</th>
<th>Rel. Freq.</th>
<th>CPI</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>0.28</td>
<td>5</td>
<td>1.40</td>
</tr>
<tr>
<td>Store</td>
<td>0.15</td>
<td>4</td>
<td>0.60</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>0.44</td>
<td>4</td>
<td>1.76</td>
</tr>
<tr>
<td>Branch</td>
<td>0.09</td>
<td>3</td>
<td>0.27</td>
</tr>
<tr>
<td>Other</td>
<td>0.04</td>
<td>4</td>
<td>0.16</td>
</tr>
<tr>
<td>Sum</td>
<td>1.00</td>
<td></td>
<td>4.19</td>
</tr>
</tbody>
</table>

$ET(Pgm) = \frac{#I \cdot CPI}{Rate} = \frac{7.5 \times 10^9 \cdot 4.19}{600 \times 10^6} = 52.375 \text{ sec}$

- Slowest CPU to execute in 35sec?

$$35 = \frac{7.5 \times 10^9 \cdot 4.19}{x}$$

$x = 898\text{MHz}$
Example 1 (cont’)

- Optimizer reduces instruction counts as follows:
- Effect on performance?

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>0.28</td>
<td>0.88</td>
<td>0.2464</td>
<td>0.2839</td>
</tr>
<tr>
<td>Store</td>
<td>0.15</td>
<td>0.96</td>
<td>0.1440</td>
<td>0.1659</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>0.44</td>
<td>0.80</td>
<td>0.3520</td>
<td>0.4056</td>
</tr>
<tr>
<td>Branch</td>
<td>0.09</td>
<td>0.95</td>
<td>0.0855</td>
<td>0.0985</td>
</tr>
<tr>
<td>Other</td>
<td>0.04</td>
<td>1.00</td>
<td>0.0400</td>
<td>0.0461</td>
</tr>
<tr>
<td>Sum</td>
<td>1.00</td>
<td></td>
<td>0.8679</td>
<td>1.0000</td>
</tr>
</tbody>
</table>

\[ #I = 0.8679 \times 7.5 \times 10^9 = 6.50925 \times 10^9 \]
Example 1 (cont’)

<table>
<thead>
<tr>
<th>Group</th>
<th>New Rel. Freq.</th>
<th>CPI</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>0.2839</td>
<td>5</td>
<td>1.4190</td>
</tr>
<tr>
<td>Store</td>
<td>0.1659</td>
<td>4</td>
<td>0.6636</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>0.4056</td>
<td>4</td>
<td>1.6224</td>
</tr>
<tr>
<td>Branch</td>
<td>0.0985</td>
<td>3</td>
<td>0.2955</td>
</tr>
<tr>
<td>Other</td>
<td>0.0461</td>
<td>4</td>
<td>0.1844</td>
</tr>
<tr>
<td>Sum</td>
<td>1.00</td>
<td>4</td>
<td>4.1849</td>
</tr>
</tbody>
</table>

\[
ET(Pgm) = \frac{\text{#I} \times \text{CPI}}{\text{Rate}} = \frac{6.50925 \times 10^9 \times 4.1849}{600 \times 10^6} = 45.401 \text{ sec}
\]

So, how much faster is the optimized code?

\[
\frac{\text{Perf(optimized)}}{\text{Perf(unoptimized)}} = \frac{\text{ET(unoptimized)}}{\text{ET(optimized)}} = \frac{52.375}{45.401} = 1.15
\]
Speeding Up Execution Time

- Execution Time is a function of three things:
  - # of instructions
  - Average CPI
  - Clock rate

- We can improve it by:
  - Choosing the “best” instruction sequence (compiler)
  - Reducing CPI (architecture)
  - Increasing clock rate (technology)

- But changing one can adversely affect the others!
Speeding Up Execution Time

- Compiler technology is quite good at generating sequences with the fewest instructions
  - Recall that this may not mean the fewest clock cycles
- Adoption of RISC architectures has led to significant reductions in average CPI
  - By using simple instructions that lend themselves to fast implementation
  - At a cost of more instructions
- Clock rates have risen between 2 and 3 orders of magnitude:
  - MIPS R2000 ca. 1985: 8MHz
  - MIPS R16000 ca. 2002: 1GHz
- Rate of change is slowing down!
Example 2

<table>
<thead>
<tr>
<th>Op</th>
<th>Freq</th>
<th>CPI&lt;sub&gt;i&lt;/sub&gt;</th>
<th>Freq x CPI&lt;sub&gt;i&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>50%</td>
<td>1</td>
<td>.5</td>
</tr>
<tr>
<td>Load</td>
<td>20%</td>
<td>5</td>
<td>1.0</td>
</tr>
<tr>
<td>Store</td>
<td>10%</td>
<td>3</td>
<td>.3</td>
</tr>
<tr>
<td>Branch</td>
<td>20%</td>
<td>2</td>
<td>.4</td>
</tr>
<tr>
<td><strong>Σ</strong></td>
<td></td>
<td><strong>2.2</strong></td>
<td></td>
</tr>
</tbody>
</table>

|         | .5   | .5            | .25                     |
| Branch  | .4   | 1.0           | 1.0                     |
| Store   | .3   | .3            | .3                      |
| ALU     | .4   | .2            | .4                      |

- How much faster would the machine be if a better data cache reduced the average load time to 2 cycles?
  
  CPU time new = 1.6 x #l x CC so 2.2/1.6 means 37.5% faster

- How does this compare with using branch prediction to shave a cycle off the branch time?
  
  CPU time new = 2.0 x #l x CC so 2.2/2.0 means 10% faster

- What if two ALU instructions could be executed at once?
  
  CPU time new = 1.95 x #l x CC so 2.2/1.95 means 12.8% faster
Which Programs Should We Analyze?

- **Kernels: Livermore loops, LINPACK**
  - Small programs that are easy to implement across architectures
  - Capture heart of a class of computations

- **Synthetic programs: Whetstone, Dhrystone**
  - Don’t perform any meaningful computation
  - But represent a model of what goes on in real computations

- **Benchmark programs: SPEC, TPC**
  - A collection of programs that represent what users do

- **Actual applications**
  - Meaningful to you
  - May not port to all systems
  - May require large data sets
Sidebar: A (Real) Anecdote

- Without optimizer: 5.2 seconds
- With optimizer: 0.1 seconds!!
- Optimizer determined that \textit{sum} was never used and therefore didn’t generate any code to compute it!

```c
void main()
{
    int i;
    double x, sum;
    sum = 0.0;
    for (i=0; i<10000; i++) {
        x = (double)i;
        sum += sqrt(x);
    }
}
```

Printing \textit{sum} resulted in optimized version running twice as fast as the unoptimized code

Moral: Understand what is going on
SPEC

- An industry consortium
  - System Performance Evaluation Cooperative
- Series of benchmarks consisting of real programs:
  - Computation-intensive
  - Graphics
  - Web servers
  - Java client/server
  - SIP
  - Virtualization
  - …
- Evolve over time:
  - CPU92 → CPU95 → CPU2000 → CPU2006
SPEC CPU2006

- SPEC CPU2006
  - CINT2006: 12 integer-only programs (C and C++)
  - CFP2006: 17 floating point programs (FORTRAN, C and C++)
- Detailed benchmark specification for reproducibility:
  - Detailed hardware characteristics
    - # of CPUs and their clock rates
    - Memory size, …
  - Detailed software characteristics:
    - Operating system version
    - Compiler used
    - Flag setting during compilation, …
- Results are relative to a baseline: Sun Ultra Enterprise 2
  - Ratio of measured wall clock time to baseline
  - A larger number is better
- Rating is a geometric mean of the individual results
- Great, but suffers from the “No child left behind” syndrome
  - Relevance
  - Manipulation
### SPEC CPU2006 Components

**CINT2006**

<table>
<thead>
<tr>
<th>Program</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>perlbench</td>
<td>Several Perl apps</td>
</tr>
<tr>
<td>bzip2</td>
<td>Data compression</td>
</tr>
<tr>
<td>gcc</td>
<td>C compiler</td>
</tr>
<tr>
<td>mcf</td>
<td>Combinatorial optimization</td>
</tr>
<tr>
<td>gobmk</td>
<td>Go program</td>
</tr>
<tr>
<td>hmmer</td>
<td>Protein sequence analysis</td>
</tr>
<tr>
<td>sjeng</td>
<td>Chess program</td>
</tr>
<tr>
<td>libquantum</td>
<td>Simulates a quantum computer</td>
</tr>
<tr>
<td>h264ref</td>
<td>Video compression</td>
</tr>
<tr>
<td>omnetpp</td>
<td>Campus-wide ethernet simulation</td>
</tr>
<tr>
<td>astar</td>
<td>Path finding algorithms</td>
</tr>
<tr>
<td>xalancbmk</td>
<td>XML processor</td>
</tr>
</tbody>
</table>

**CFP2006**

<table>
<thead>
<tr>
<th>Program</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>bwaves</td>
<td>Fluid dynamics</td>
</tr>
<tr>
<td>gamess</td>
<td>Quantum chemistry</td>
</tr>
<tr>
<td>milc</td>
<td>Quantum chromodynamics</td>
</tr>
<tr>
<td>zeusmp</td>
<td>Computational fluid dynamics</td>
</tr>
<tr>
<td>gromacs</td>
<td>Molecular dynamics</td>
</tr>
<tr>
<td>cactusADM</td>
<td>General relativity</td>
</tr>
<tr>
<td>leslie3D</td>
<td>Computational fluid dynamics</td>
</tr>
<tr>
<td>namd</td>
<td>Molecular dynamics</td>
</tr>
<tr>
<td>dealII</td>
<td>Finite element analysis</td>
</tr>
<tr>
<td>soplex</td>
<td>Simplex algorithm</td>
</tr>
<tr>
<td>povray</td>
<td>Image ray tracing</td>
</tr>
<tr>
<td>calculix</td>
<td>Structural mechanics</td>
</tr>
<tr>
<td>GemsFDTD</td>
<td>Computational electromagnetics</td>
</tr>
<tr>
<td>tonto</td>
<td>Quantum chemistry</td>
</tr>
<tr>
<td>lbm</td>
<td>Fluid dynamics</td>
</tr>
<tr>
<td>wrf</td>
<td>Weather modeling</td>
</tr>
<tr>
<td>sphinx3</td>
<td>Speech recognition</td>
</tr>
</tbody>
</table>
TPC: Transaction Processing Benchmarks

- Define an application scenario
  - Involves end-users
  - Remote access over a network
  - Databases

- Considers throughput, latency, and price

- Originally developed for ATM-like transactions

- Now focused on order-entry application developed at MCC in mid-80’s

- Attempt at reality and completeness
  - But at the price of tremendous complexity
Performance Speed Up

SpeedUp = \frac{ET \text{ Before Change}}{ET \text{ After Change}}

Speedup depends on:
Goodness of enhancement (s)
Fraction of time it’s used (p)

ET After Change = ET \text{ Before Change} \times \left[ (1 - p) + \frac{p}{s} \right]
Amdahl’s Law

- Gene Amdahl: IBM S/360 Chief Architect

\[
\text{Speedup} = \frac{1}{(1 - p) + \frac{p}{s}}
\]

- Speedup bounded by: \( \frac{1}{\text{Fraction of time not enhanced}} \)

- Duh!
Example 3

- Can double performance of floating point instructions
- Cut latency by a factor of 2
- Floating point operations represent 10% of workload

\[ \text{ETAfter} = \text{ETBefore} \times \left[ 0.9 + \frac{0.1}{2} \right] = 0.95 \times \text{ETBefore} \]

\[ \text{SpeedUp} = \frac{1}{0.95} = 1.053 \]
Example 4

- Application takes 100 sec to run
- Multiplication represents 80% of the work
- How much faster would multiplication have to be in order to get performance to improve by a factor of 4?

\[ 25 = 100 \times \left[ 0.2 + \frac{0.8}{x} \right] \]

- How about a factor of 6?
Amdahl’s Law

- Make the common case fast!
- Performance improvement depends on:
  - Goodness of the enhancement
  - Frequency of use
- Examples
  - All instructions require instruction fetch, only a fraction require data
    - Optimize instruction access first
  - Programs exhibit data locality; small memories are faster
    - Storage hierarchy: most frequent access to small, fast, local memory
Summary: Evaluating ISAs

- **Design-time metrics:**
  - Can it be implemented, in how long, at what cost?
  - Can it be programmed? Ease of compilation?

- **Static Metrics:**
  - How many bytes does the program occupy in memory?

- **Dynamic Metrics:**
  - How many instructions are executed? How many bytes does the processor fetch to execute the program?
  - How many clocks are required per instruction?
  - How fast can the clock be made?

*Best Metric:* Time to execute the program!

depends on the instructions set, the processor organization, and compilation techniques.
Beauty is in the Eye of the Beholder

The right metric depends on the application:
- Desktop
- Game console
- Microwave oven microcontroller
- Web server

The right metric depends on the perspective:
- CPU designer
- System architect
- Customer

Opportunity for manipulation galore!
Our implementation of the MIPS is simplified:
- Memory-reference instructions: \texttt{lw}, \texttt{sw}
- Arithmetic-logical instructions: \texttt{add}, \texttt{sub}, \texttt{and}, \texttt{or}, \texttt{slt}
- Control flow instructions: \texttt{beq}, \texttt{j}

Generic implementation:
- Use the program counter (PC) to supply the instruction address and fetch the instruction from memory (and update the PC)
- Decode the instruction (and read registers)
- Execute the instruction

Later - more realistic pipelined version
- All instructions (except \texttt{j}) use the ALU after reading the registers
Instruction Execution

- PC $\rightarrow$ instruction memory, fetch instruction
- Register numbers $\rightarrow$ register file, read registers
- Depending on instruction class
  - Use ALU to calculate
    - Arithmetic result
    - Memory address for load/store
    - Branch target address
  - Access data memory for load/store
- PC $\leftarrow$ target address or PC + 4
CPU Overview
Multiplexers

- Can’t just join wires together
- Use multiplexers
Control
Logic Design Basics

- Information encoded in binary
  - Low voltage = 0, High voltage = 1
  - One wire per bit
  - Multi-bit data encoded on multi-wire buses

- Combinational element
  - Operate on data
  - Output is a function of input

- State (sequential) elements
  - Store information
Combinational Elements

- **AND-gate**
  - \( Y = A \& B \)

- **Multiplexer**
  - \( Y = S \ ? \ I1 \ : \ I0 \)

- **Adder**
  - \( Y = A + B \)

- **Arithmetic/Logic Unit**
  - \( Y = F(A, B) \)
Sequential Elements

- Register: stores data in a circuit
  - Uses a clock signal to determine when to update the stored value
  - Edge-triggered: update when Clk changes from 0 to 1
Sequential Elements

- Register with write control
  - Only updates on clock edge when write control input is 1
  - Used when stored value is required later

![Diagram of a register with write control](image)
Clocking Methodologies

- The clocking methodology defines when signals can be read and when they are written
  - An edge-triggered methodology
  - Longest delay determines clock period
- Typical execution
  - read contents of state elements
  - send values through combinational logic
  - write results to one or more state elements

Assumes state elements are written on every clock cycle; if not, need explicit write control signal

write occurs only when both the write control is asserted and the clock edge occurs
Building a Datapath

- Datapath
  - Elements that process data and addresses in the CPU
    - Registers, ALUs, mux’s, memories, …

- We will build a MIPS datapath incrementally
  - Refining the overview design
Abstract Implementation View

- Two types of functional units:
  - elements that operate on data values (combinational)
  - elements that contain state (sequential)

Single cycle operation
Split memory model - one memory for instructions and one for data
Instruction Fetch

- Fetching instructions involves
  - reading the instruction from the Instruction Memory
  - updating the PC to hold the address of the next instruction

PC is updated every cycle, so it does not need an explicit write control signal
Instruction Memory is read every cycle, so it doesn’t need an explicit read control signal
Decoding instructions involves

- sending the fetched instruction’s opcode and function field bits to the control unit

reading two values from the Register File
Register File addresses are contained in the instruction
R Format Instructions

- **R format operations** *(add, sub, slt, and, or)*

```
31  25  20  15  10  5  0
```

- **R-type**: `{op} rs rt rd shamt funct`

- perform the *(op and funct)* operation on values in *rs* and *rt*
- store the result back into the Register File (into location *rd*)

The Register File is not written every cycle (e.g. *sw*), so we need an explicit write control signal for the Register File
Load and Store Instructions

- Load and store operations involve:
  - Read register operands
  - Compute memory address by adding the base register to the 16-bit signed-extended offset field in the instruction
  - **Store** value (read from the Register File) written to the Data Memory
  - **Load** value, read from the Data Memory, written to the Register File
Composing the Elements

- First-cut data path does an instruction in one clock cycle
  - Each datapath element can only do one function at a time
  - Hence, we need separate instruction and data memories
- Use multiplexers where alternate data sources are used for different instructions
R-Type/Load/Store Datapath
Branch Instructions

- Read register operands
- Compare operands
  - Use ALU, subtract and check Zero output
- Calculate target address
  - Sign-extend displacement
  - Shift left 2 places (word displacement)
  - Add to PC + 4
    - Already calculated by instruction fetch
Branch Instructions

- Branch operations involve:
  - compare the operands read from the Register File during decode for equality (**zero** ALU output)
  - compute the branch target address by adding the updated PC to the 16-bit signed-extended offset field in the instruction

```
Branch Instructions

- Branch operations involve:
  - compare the operands read from the Register File during decode for equality (**zero** ALU output)
  - compute the branch target address by adding the updated PC to the 16-bit signed-extended offset field in the instr

Branch target address
```

```
Instruction

Add

PC

4

Add

Branch target address

Shift left 2

Add

zero (to branch control logic)

Add

ALU operation

Read Addr 1

Read Addr 2

Read Data 1

Read Data 2

Write Addr

Write Data

Register File

ALU

16

32

Sign Extend

4

PC

Read Addr 1

Read Addr 2

Read Data 1

Read Data 2

Write Addr

Write Data

Register File

ALU

16

32

Sign Extend

4

PC

Branch target address
```

University of Texas at Austin   CS352H   Computer Systems Architecture   Fall 2009   Don Fussell   48
Jump Instruction

- Jump operation involves
  - replace the lower 28 bits of the PC with the lower 26 bits of the fetched instruction shifted left by 2 bits

![Diagram of jump instruction process](image)
Creating a Single Datapath from the Parts

- Assemble the datapath segments and add control lines and multiplexors as needed
- **Single cycle** design – fetch, decode and execute each instruction in **one** clock cycle
  - no datapath resource can be used more than once per instruction, so some must be duplicated (e.g., separate Instruction Memory and Data Memory, several adders)
  - **multiplexors** needed at the input of shared elements with control lines to do the selection
  - write signals to control writing to the Register File and Data Memory
- **Cycle time is determined by length of the longest path**
Fetch, R, and Memory Access Portions

- Fetch
- Read Address
- Instruction
- Add
- PC
- Instruction Memory
- Read Address
- Instruction
- Write Data
- Read Addr 1
- Read Addr 2
- Read Data 1
- Read Data 2
- Register
- File
- ALU
- RegWrite
- ALU control
- MemWrite
- Address
- Data Memory
- Write Data
- Sign Extend
- 16
- Extend
- 32
Multiplexor Insertion

- Instruction Memory
  - Read Address
  - Read Data
- Register File
  - Read Addr 1
  - Read Addr 2
  - Write Addr
  - Write Data
- ALU
  - ALUSrc
  - ALU control
- ALU output
  - ovf
  - zero
- Memory
  - Read Data
  - Write Data
- Sign Extend
  - 16
  - 32
- MemRead
  - Write Data
  - Address
- MemWrite
  - MemtoReg
  - RegWrite
Clock Distribution

System Clock

(clock cycle)

Instruction Memory
- Read Address
- Read Instruction

Register File
- Read Addr 1
- Read Addr 2
- Read Data
- Write Addr
- Write Data

Adder
- 4

ALU
- ovf
- zero
- ALUSrc
- ALU control

Data Memory
- Address
- Read Data
- Write Data
- MemRead

Register
- RegWrite

Sign Extension
- 16
- 32
Adding the Branch Portion
Full Datapath
ALU Control

- ALU used for
  - Load/Store: F = add
  - Branch: F = subtract
  - R-type: F depends on funct field

<table>
<thead>
<tr>
<th>ALU control</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>AND</td>
</tr>
<tr>
<td>0001</td>
<td>OR</td>
</tr>
<tr>
<td>0010</td>
<td>add</td>
</tr>
<tr>
<td>0110</td>
<td>subtract</td>
</tr>
<tr>
<td>0111</td>
<td>set-on-less-than</td>
</tr>
<tr>
<td>1100</td>
<td>NOR</td>
</tr>
</tbody>
</table>
## ALU Control

- Assume 2-bit ALUOp derived from opcode
- Combinational logic derives ALU control

<table>
<thead>
<tr>
<th>opcode</th>
<th>ALUOp</th>
<th>Operation</th>
<th>funct</th>
<th>ALU function</th>
<th>ALU control</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>00</td>
<td>load word</td>
<td>XXXXXX</td>
<td>add</td>
<td>0010</td>
</tr>
<tr>
<td>sw</td>
<td>00</td>
<td>store word</td>
<td>XXXXXX</td>
<td>add</td>
<td>0010</td>
</tr>
<tr>
<td>beq</td>
<td>01</td>
<td>branch equal</td>
<td>XXXXXX</td>
<td>subtract</td>
<td>0110</td>
</tr>
<tr>
<td>R-type</td>
<td>10</td>
<td>add</td>
<td>100000</td>
<td>add</td>
<td>0010</td>
</tr>
<tr>
<td></td>
<td></td>
<td>subtract</td>
<td>100010</td>
<td>subtract</td>
<td>0110</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AND</td>
<td>100100</td>
<td>AND</td>
<td>0000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OR</td>
<td>100101</td>
<td>OR</td>
<td>0001</td>
</tr>
<tr>
<td></td>
<td></td>
<td>set-on-less-than</td>
<td>101010</td>
<td>set-on-less-than</td>
<td>0111</td>
</tr>
</tbody>
</table>
### The Main Control Unit

**Control signals derived from instruction**

<table>
<thead>
<tr>
<th>Type</th>
<th>Opcode</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load/Store</td>
<td>35 or 43</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>31:26</td>
<td>25:21</td>
<td>20:16</td>
<td></td>
<td>15:0</td>
<td></td>
</tr>
<tr>
<td>Branch</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>31:26</td>
<td>25:21</td>
<td>20:16</td>
<td></td>
<td>15:0</td>
<td></td>
</tr>
</tbody>
</table>

- **opcode**: always read
- **always read**: read, except for load
- **write for R-type and load**: sign-extend and add
Main Control Unit

<table>
<thead>
<tr>
<th>Instr</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>MemReg</th>
<th>RegWr</th>
<th>MemRd</th>
<th>MemWr</th>
<th>Branch</th>
<th>ALUOp</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type 000000</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>lw 100011</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>sw 101011</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>beq 000100</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>01</td>
</tr>
</tbody>
</table>
Design the Main Control logic from the truth table

- R-type
- lw
- sw
- beq

- Instr[31]
- Instr[30]
- Instr[29]
- Instr[28]
- Instr[27]
- Instr[26]

- RegDst
- ALUSrc
- MemtoReg
- RegWrite
- MemRead
- MemWrite
- Branch
- ALUOp₁
- ALUOp₀
R-Type Instruction
Load Instruction
Branch-on-Equal Instruction
Implementing Jumps

<table>
<thead>
<tr>
<th>Jump</th>
<th>2</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>31:26</td>
<td>25:0</td>
</tr>
</tbody>
</table>

- Jump uses word address
- Update PC with concatenation of
  - Top 4 bits of old PC
  - 26-bit jump address
  - 00
- Need an extra control signal decoded from opcode
## Main Control Unit

<table>
<thead>
<tr>
<th>Instr</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>MemReg</th>
<th>RegWr</th>
<th>MemRd</th>
<th>MemWr</th>
<th>Branch</th>
<th>ALUOp</th>
<th>Jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>lw</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>j</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>X</td>
<td>XX</td>
<td>1</td>
</tr>
</tbody>
</table>
Performance Issues

- Longest delay determines clock period
  - Critical path: load instruction
  - Instruction memory → register file → ALU → data memory → register file
- Not feasible to vary period for different instructions
- Violates design principle
  - Making the common case fast
- We will improve performance by pipelining
Instruction Critical Paths

- Calculate cycle time assuming negligible delays (for muxes, control unit, sign extend, PC access, shift left 2, wires) except:
  - Instruction and Data Memory (4 ns)
  - ALU and adders (2 ns)
  - Register File access (reads or writes) (1 ns)

<table>
<thead>
<tr>
<th>Instr.</th>
<th>I Mem</th>
<th>Reg Rd</th>
<th>ALU Op</th>
<th>D Mem</th>
<th>Reg Wr</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>load</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>store</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>beq</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>jump</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
</tr>
</tbody>
</table>
Single Cycle Disadvantages & Advantages

- Uses the clock cycle inefficiently – the clock cycle must be timed to accommodate the **slowest** instruction
  - especially problematic for more complex instructions like floating point multiply

```
<table>
<thead>
<tr>
<th>Clk</th>
<th>Cycle 1</th>
<th>Cycle 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>lw</td>
<td>sw</td>
</tr>
<tr>
<td></td>
<td>Waste</td>
<td></td>
</tr>
</tbody>
</table>
```

- May be wasteful of area since some functional units (e.g., adders) must be duplicated since they cannot be shared during a clock cycle but
- Is simple and easy to understand
Next Lecture

- MIPS pipelined implementation
- Rest of chapter 4