Notes and Updates

- Super quick breeze through Chapter 3
  - You learned most (all) of this in CSE140
- Homework Due Tues! (no lab Mon)
- Candy for latency/throughput people
- Homework due NO LATER than 2:05pm
  - No handing it in at the end of class
- QUESTIONS!

Do you use the web system?

- 1) In Class:
  - Yes, why?
  - No, why not?
- 2) After class:
  - Yes, why?
  - No why, not?
- 3) Before class?
Performance Beyond one Program

<table>
<thead>
<tr>
<th>Program</th>
<th>Computer A</th>
<th>Computer B</th>
<th>Computer C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program 1</td>
<td>1</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>Program 2</td>
<td>1000</td>
<td>100</td>
<td>20</td>
</tr>
<tr>
<td>Total Time</td>
<td>1001</td>
<td>110</td>
<td>40</td>
</tr>
</tbody>
</table>

- Which machine is fastest?

How to summarize performance

- Arithmetic Mean
  \[
  \frac{1}{n} \sum_{i=1}^{n} \text{Time}_i
  \]

- Weighted Arithmetic Mean
  \[
  \sum_{i=1}^{n} \text{Time}_i \times \text{Weight}_i \quad (\text{Weights total to 1})
  \]

- Harmonic Mean
  \[
  \frac{n}{\sum_{i=1}^{n} \frac{1}{\text{Rate}_i}}
  \]

- Geometric Mean
  \[
  \sqrt[n]{\prod_{i=1}^{n} \frac{\text{Execution Time Ratio}_i}{1}}
  \]
Performance Beyond one Program

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>W(1)</th>
<th>W(another)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program 1</td>
<td>1</td>
<td>10</td>
<td>20</td>
<td>.5</td>
<td>.999</td>
</tr>
<tr>
<td>Program 2</td>
<td>1000</td>
<td>100</td>
<td>20</td>
<td>.5</td>
<td>.001</td>
</tr>
<tr>
<td>AM: W(1)</td>
<td>500</td>
<td>55</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AM: W(ather)</td>
<td>2</td>
<td>10</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GM</td>
<td>31.6</td>
<td>31.6</td>
<td>20</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Summarizing Performance

- Even an “unweighted” arithmetic mean IS weighted
  - The longer the running time, the greater the impact of that code on the mean

- Geometric means of normalized execution times are consistent no matter which machine is faster
  - Ratios of geometric means always give equal weights to all benchmarks - no matter execution times

- Geometric mean does not necessarily prediction execution time for any mix of the programs
Another way of “measuring” performance:

- It’s hard to convince manufacturers to run your program (unless you’re a BIG customer)
- A benchmark is a set of programs that are representative of a class of problems.
  - measure one feature of system
    - e.g. memory accesses or communication speed
  - most compute-intensive part of applications
    - e.g. Linpack and NAS kernel b’marks (for supercomputers)
  - Full application:
    - (int and float) (for Unix workstations)
    - Other suites for databases, web servers, graphics,...

SPEC89 and the compiler

Darker bars show performance with compiler improvements (same machine as light bars)
SPEC on Pentium III and Pentium 4

- What do you notice?

Other SPECs

- HPC (High Performance Computing)
  - Quantum Chemistry, Weather Modeling, Seismic
- JVM (Java)
- JAppletServer
- Web
- Mail
- JBB Java Business Benchmark
- SFS System File Server

Test many things other than the CPU speed - test entire system performance
Performance Beyond the CPU

- We (and this book) concentrate on the CPU as a lone entity
  - For a while (Chap 7,8,9)
- Memory: A very important part
  - The CPU can only do work if it has data to work on
  - Latency and Bandwidth were our metrics
- Due to modern processor design, improving speed of integer operations by 10% will (likely) NOT speed up ANYTHING!

Key Points

- Be careful how you specify performance
  - Use times faster, practice!
- Execution time = instructions *CPI *cycle time
- Make the common case FAST!
  - Amdahl's Law
- Use real applications to measure performance
  - Make sure their workload represents the one you care about!
- Use geometric mean to report performance on suites of programs or benchmarks
Chapter 4: Arithmetic for Computers

NOTE: Much of this material you should already know from CSE140 (up through 3.5)

*THIS IS JUST A REVIEW*
Binary Numbers

Consider a 4-bit binary number

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>6</td>
<td>0110</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>7</td>
<td>0111</td>
</tr>
</tbody>
</table>

Examples of binary arithmetic:

\[
3 + 2 = 5 \\
3 + 3 = 6
\]

\[
\begin{array}{cccc}
0 & 0 & 1 & 1 \\
+ & 0 & 0 & 1 & 0 \\
\end{array}
\]

\[
\begin{array}{cccc}
0 & 0 & 1 & 1 \\
+ & 0 & 0 & 1 & 1 \\
\end{array}
\]

What about negative integers?

- Desirable features of a number system ...
  - obvious representation of 0,1,2...
  - uses adder for addition
  - easy to recognize exceptions (like overflow)
  - single value of 0
  - equal coverage of positive and negative numbers
  - easy detection of sign
  - easy negation
Some Alternatives

• Sign Magnitude -- MSB is sign bit
  -1 → 1001
  -5 → 1101
• One’s complement -- flip all bits to negate
  -1 → 1110
  -5 → 1010

Two’s Complement Representation

- Positive numbers: normal binary representation
- Negative numbers: flip bits (0 ↔ 1), then add 1

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Two’s Complement Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>-8</td>
<td>1000*</td>
</tr>
<tr>
<td>-7</td>
<td>1001</td>
</tr>
<tr>
<td>-6</td>
<td>1010</td>
</tr>
<tr>
<td>-5</td>
<td>1011</td>
</tr>
<tr>
<td>-4</td>
<td>1100</td>
</tr>
<tr>
<td>-3</td>
<td>1101</td>
</tr>
<tr>
<td>-2</td>
<td>1110</td>
</tr>
<tr>
<td>-1</td>
<td>1111</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
</tr>
<tr>
<td>7</td>
<td>0111*</td>
</tr>
</tbody>
</table>

Smallest 4-bit number: -8
Biggest 4-bit number: 7
Two's Complement Arithmetic:
So cool for adders
Uses simple adder for + and - numbers

\[
\begin{align*}
7 + (-6) &= 1 \\
3 + (-5) &= -2
\end{align*}
\]

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

Arithmetic -- The heart of instruction execution
A One Bit ALU

- This 1-bit ALU will perform AND, OR, and ADD
**The Disadvantage of Ripple Carry**

- **Simple Adders are Ripple Carry**
  - The carry bit may have to propagate from LSB to MSB
  - Worst case delay for an N-bit RC adder:

```
A0  B0  CarryIn0  1-bit ALU  Result0  CarryOut0
A1  B1  CarryIn1  1-bit ALU  Result1  CarryOut1
A2  B2  CarryIn2  1-bit ALU  Result2  CarryOut2
A3  B3  CarryIn3  1-bit ALU  Result3  CarryOut3
```

**A Partial Carry Lookahead Adder**

- It is very expensive to build a "full" carry lookahead adder
  - Just imagine the length of the equation for Cin31
- Common practices:
  - Connect several N-bit Lookahead Adders to form a big adder
  - Example: connect four 8-bit carry lookahead adders to form a 32-bit partial carry lookahead adder

```
  /\   \
  8     8      8     8      8     8      8     8

8-bit Carry Lookahead Adder   8-bit Carry Lookahead Adder   8-bit Carry Lookahead Adder   8-bit Carry Lookahead Adder
8     8     8     8     8     8     8
```

*Worst-case delay??*
Key Points

- Two’s complement is standard +/- numbers.
  - Achieves almost all of our goals.
- CPU clock speed is driven by adder delay (and mult and div)
  - Adder is used in loads, stores and branches as well as arithmetic.
  - Thus, using a carry-lookahead adder is important!

Chapter 5:
The Processor: Datapath and Control
The Single Cycle Processor
The Multicycle Processor

Note: Some of the material in this lecture are COPYRIGHT 1998 MORGAN KAUFMANN PUBLISHERS, INC. ALL RIGHTS RESERVED.
Figures may be reproduced only for classroom or personal education use in conjunction with our text and only when the above line is included.
The Performance Big Picture

- Execution Time = Instrs * CPI * Cycle Time
- Processor design (datapath and control) will determine:
  - Clock cycle time
  - Clock cycles per instruction

- Starting today:
  - Single cycle processor:
    - Advantage: CPI = 1
    - Disadvantage: long cycle time

What parts of MIPS?

- We won't implement all of MIPS
  - Memory instructions
  - Arithmetic/Logical (and just a subset of these, but you should be able to figure out how to add many of them)
  - BEQ and J (last)
- Basic load/store architecture with these steps:
  - Read PC and Fetch Inst
  - Read Registers
  - Do Math
  - Write memory/registers
  - Repeat
- Graphically?
Basics (page 287 minus a few things) of Single Cycle Datapath Design

What’s Datapath? What’s Control?

Processor Design: Logic Components, Time, the Clock

- Review: Section 5.2
- Registers, Memory - these things we need to get values out of and write new values into
  - Based on the clock cycle
- Our cycle will be based around the rising clock edge
  - Set values to be ready for that edge, and then a read or write will happen at that edge
  - Do your work to calculate what to read or write in the “rest” of the cycle
Processor Design

- We're ready to implement the MIPS “core”
  - load-store instructions: lw, sw
  - reg-reg instructions: add, sub, and, or, slt
  - control flow instructions: beq
- First, we need to fetch an instruction into processor
  - supplies instruction address
  - get the instruction from memory

That was too easy

- A problem – how will we do a load or store?
Instruction & Data in same cycle?
Solution: separate data and instruction memory
There will be only one DRAM memory
We want a stored program architecture
How else can you compile and then run a program??
But we can have separate SRAM caches
(We'll study caches later)

Instruction Fetch Unit
Updating the PC for next instruction
- Sequential Code:
- Branch and Jump:
  • We'll save branches for later, after adds, subs
The MIPS core subset

- **R-type**
  - `add rd, rs, rt`
  - `sub, and, or, slt`

- **LOAD and STORE**
  - `lw rt, rs, imm`
  - `sw rt, rs, imm`

- **BRANCH:**
  - `beq rs, rt, imm`

The following table summarizes the fields and their functionalities:

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>6 bits</td>
<td>0-61116212631</td>
</tr>
<tr>
<td>rs</td>
<td>5 bits</td>
<td>registers</td>
</tr>
<tr>
<td>rt</td>
<td>5 bits</td>
<td>rt</td>
</tr>
<tr>
<td>rd</td>
<td>5 bits</td>
<td>rd</td>
</tr>
<tr>
<td>shamt</td>
<td>5 bits</td>
<td>shamt</td>
</tr>
<tr>
<td>funct</td>
<td>6 bits</td>
<td>funct</td>
</tr>
</tbody>
</table>

**Example Instructions**

- Read registers `rs` and `rt` for `add rd, rs, rt`
- Feed `rs` and immediate to ALU for `lw rt, rs, imm`
- Move data between memory and register for `sw rt, rs, imm`

---

Register Transfer Language (RTL)

- **Is a mechanism for describing the movement of data between storage elements**
- **Gives us a precise way to describe various actions of our instructions**
  - May be more than 1 RTL statement per instruction
- `PC <= PC + 4`
- `R[rd] <= R[rs] + R[rt]`
Post Fetch Datapath for Reg-Reg Operations

- \( R[rd] \leftarrow R[rs] \text{ op } R[rt] \) Example: \( add \ rd, rs, rt \)
  - \( Ra(1), Rb(2), \) and \( Rw \) come from \( rs, rt, \) and \( rd \) fields
  - \( ALU \) operation signal depends on \( op \) and \( funct \)

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>26</th>
<th>21</th>
<th>16</th>
<th>11</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>( op )</td>
<td>6 bits</td>
<td>( rs )</td>
<td>5 bits</td>
<td>( rt )</td>
<td>5 bits</td>
<td>( rd )</td>
<td>5 bits</td>
</tr>
</tbody>
</table>

Post Fetch Datapath for Store Operations

\( Mem[R[rs] + \text{SignExt}[imm16]] \leftarrow R[rt] \)
Example: \( sw \ rt, rs, imm16 \)
Putting together Store DP and RR DP

Post Fetch Datapath for Load Operations

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

Example: \( lw \ rt, rs, \text{imm16} \)
Putting together Load/Store DP and Reg-Reg DP

Datapath for Branch Operations

`beq rs, rt, imm16` We need to compare Rs and Rt
Computing the Next Address

- PC is a 32-bit byte address into the instruction memory
  - Sequential operation: $PC_{31:0} = PC_{31:0} + 4$
  - Branch: $PC_{31:0} = PC_{31:0} + 4 + \text{SignExt}[\text{Imm16}] \times 4$
- We don’t need the 2 least-significant bits because:
  - The 32-bit PC is a byte address
  - And all our instructions are 4 bytes (32 bits) long
  - The 2 LSB’s of the 32-bit PC are always zeros

Detour:
Multiply -- That’s expensive!
- Multiply the immediate by 4! Let’s try some possible values

```
0000 0001
0000 0010
0000 0011
0000 0100
1111 1111
```
Datapath for Branch Operations

```
beq rs, rt, imm16 We need to compare Rs and Rt
```

![Datapath diagram]

All together: the single cycle datapath

```
Add ALU rs
Shift left 2
Zero
```

![Datapath diagram]
The R-Format (e.g. *add*) Datapath

ALUsrc  ALUop  Mem  Read  MemWrite  MemToReg  RegDst  RegWrite  PCsrc

The Load Datapath

ALUsrc  ALUop  Mem  Read  MemWrite  MemToReg  RegDst  RegWrite  PCsrc
The Store Datapath

The beq Datapath
Key Points

- CPU is just a collection of state and combinational logic
- We just designed a very rich processor, at least in terms of functionality
  - Know and understand
    - Basic flow
    - Control lines
    - Muxes - where and why needed
- Execution time = Insts * CPI * Cycle Time
  - where does the single-cycle machine fit in?

Adding Control Signals
DETOUR: Single Cycle Datapath

Warning! Text is inconsistent. MUX control signals sometimes have "1" is on top, sometimes "0". On exercises&tests, look carefully!

Control for instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>MemtoReg</th>
<th>Reg Write</th>
<th>Mem Read</th>
<th>Mem Write</th>
<th>Branch</th>
<th>ALUOp1</th>
<th>ALUOp0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R format</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>beq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>j</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jal</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>br branch</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bcond</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>j</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

54