Review

Measuring performance
Impact of branches, cache misses
Reporting performance comparisons
  Arithmetic and weighted means
  Relative performance ratios
  Geometric mean
Benchmarks, kernels and synthetic benchmarks
Mean time to failure and mean time to repair (MTTF/MTTR)

Amdahl’s law:
  for single core- performance is limited by the fraction that is improved
  for parallel systems: performance is limited by serial fraction

Homework #1   1.3, 1.7, 1.10, 1.14
Due: Jan 31, 2017

Instruction Set Architecture mostly from Appendix A
How should we design an instruction set?
Difference between Assembly language and Machine language (Instruction set)

Instruction Format
  How many bits per instruction
  How many bits for opcode
  What and how many operands to include

<table>
<thead>
<tr>
<th>OPCODE</th>
<th>Operand-1</th>
<th>Operand-2</th>
<th>Operand-3</th>
<th>...</th>
<th>Operand-n</th>
</tr>
</thead>
</table>

Book classifies the ISA into 4 types (see page A-4)
Stack
Accumulator
Register-Memory
Load/store (register-register)
Let us see how these ISA's differ with a simple example
Convert $C = A + B$ into these instruction sets

<table>
<thead>
<tr>
<th></th>
<th>Stack</th>
<th>Accumulator</th>
<th>Register-Memory</th>
<th>Load/Store (Reg-Reg)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUSH A</td>
<td>LOAD A</td>
<td>LOAD R1, A</td>
<td>LOAD R1, A</td>
<td></td>
</tr>
<tr>
<td>PUSH B</td>
<td>ADD B</td>
<td>ADD R3, R1, B</td>
<td>LOAD R2, B</td>
<td></td>
</tr>
<tr>
<td>ADD</td>
<td>STORE C</td>
<td>STORE C, R3</td>
<td>ADD R3, R1, R2</td>
<td></td>
</tr>
<tr>
<td>POP C</td>
<td></td>
<td>STORE C, R3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Which is better?
We can try to compare code sizes.
Let us look at problem A.9 from Text on page A-49

A.9 [10/15] <A.2> For the following assume that values A, B, C, D, E, and F reside in memory. Also assume that instruction operation codes are represented in 8 bits, memory addresses are 64 bits, and register addresses are 6 bits.

a. [10] <A.2> For each instruction set architecture shown in Figure A.2, how many addresses, or names, appear in each instruction for the code to compute $C = A + B$, and what is the total code size?

b. [15] <A.2> Some of the instruction set architectures in Figure A.2 destroy operands in the course of computation. This loss of data values from processor internal storage has performance consequences. For each architecture in Figure A.2, write the code sequence to compute:

$C = A + B$
$D = A - E$
$F = C + D$

In your code, mark each operand that is destroyed during execution and mark each "overhead" instruction that is included just to overcome this loss of data from processor internal storage. What is the total code size, the number of bytes of instructions and data moved to or from memory, the number of overhead instructions, and the number of overhead data bytes for each of your code sequences?
But implementing Stack instructions is very difficult using pipelines. Accumulator may require multiple loads.

Consider \( C = A + B + A \times D \)

<table>
<thead>
<tr>
<th>Stack</th>
<th>Accumulator</th>
<th>Register-Memory</th>
<th>Load/Store (Reg-Reg)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUSH A (8+64)</td>
<td>LOAD A (8+64)</td>
<td>LOAD R1, A (8+6+64)</td>
<td>LOAD R1, A (8+6+64)</td>
</tr>
<tr>
<td>PUSH B (8+64)</td>
<td>ADD B (8+64)</td>
<td>ADD R3, R1, B (8+6+6+6+64)</td>
<td>LOAD R2, B (8+6+64)</td>
</tr>
<tr>
<td>ADD (8)</td>
<td>STORE C (8+64)</td>
<td>STORE C, R3 (8+6+4+6)</td>
<td>ADD R3, R1, R2 (8+6+6+6+6+6)</td>
</tr>
<tr>
<td>POP C (8+64)</td>
<td></td>
<td>STORE C, R3 (8+64+6)</td>
<td></td>
</tr>
</tbody>
</table>

224 bits 216 bits 240 bits 260 bits

See page A6 for a discussion of Pros and cons on these different ISAs.
CSCE 4610: Computer Architecture

How is this represented in memory if we are using 32 bit words?
Note we need to align certain items on appropriate byte boundaries

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>word 0</td>
<td>a</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>word 1</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td>word 2</td>
<td>c</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>word 3</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
</tr>
<tr>
<td>word 4</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
</tr>
<tr>
<td>word 5</td>
<td>e</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>word 6</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
</tr>
<tr>
<td>word 7</td>
<td>g</td>
<td>g</td>
<td>g</td>
<td>g</td>
</tr>
<tr>
<td>word 8</td>
<td>g</td>
<td>g</td>
<td>g</td>
<td>g</td>
</tr>
<tr>
<td>word 9</td>
<td>cptr</td>
<td>cptr</td>
<td>cptr</td>
<td>cptr</td>
</tr>
<tr>
<td>word 10</td>
<td>fptr</td>
<td>fptr</td>
<td>fptr</td>
<td>fptr</td>
</tr>
<tr>
<td>word 11</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

How does it look like if we are using 64 bit words?

We can rearrange the elements of the struct to save some bytes

CSCE 4610: Computer Architecture

Address modes (page A-9) – how do we specify memory address in a instruction

Register, immediate -- not really address, data is in a register
ADD R_i, R_j, R_k

Register indirect – address is in a register
Load R_i, (R_j)

Immediate -- “data” is in the instruction itself
ADDI R_i, R_j, #4

Displacement, Indexed – address is the sum of the displacement and the value in a registers:
LOAD R_i, 100(R_j)

Absolute (or memory direct) – address specified in the instruction
LOAD R_i, 10000

Memory Indirect – address in the instruction points to memory location that actually contains the address:
LOAD R_i, (10000)

Auto increment and decrement

How to specify which mode is being used?
CSCE 4610: Computer Architecture

Separate field for mode

<table>
<thead>
<tr>
<th>OPCODE</th>
<th>Mode</th>
<th>Operand-1</th>
<th>Mode</th>
<th>Operand-2</th>
</tr>
</thead>
</table>

Not good to have too many modes
- compiler analysis becomes more complex
- need more bits
- some modes make no sense with some instructions

OR the address mode can be implied by the operation
- ADD-Immediate \( R_i, R_j, #\text{value} \)
- Load-Auto-inc \( R_i, \text{disp}(R_j) \)

We allow addresses with load/store (and branch instructions)
And we limit the address mode to Register + displacement only

\[
\text{Load } R_i, \text{disp}(R_j) \\
\text{Store } R_i, \text{disp}(R_j)
\]

-- note we can make displacement = 0 (will be register indirect mode)
or use \( R_0 \), so the register value is always zero (will be absolute address)

\[
\text{Load } R_i, 0 (R_j) \\
\text{Store } R_i, \text{disp}(R_j)
\]

How large should the displacement field be?
- Need displacement for immediate values, address displacement and branches

See the figure on page A-12
It shows that you rarely need more than 16 bits of displacement for addresses
CSCE 4610: Computer Architecture

For immediate values

Other types of operations that need addresses

Branch instructions – PC relative mode

Compare and Branch

BEQ R_i, R_j, Target-Address : Similar to Load R_i, disp (R_j)
Jump

J Target-Address : New format– no registers
JAL function-address : J disp or JR (R_j)

How many bits for displacement in Jump?

CSCE 4610: Computer Architecture

See the figure on page A-18
This shows how far do you normally jump (relative to current position)

In most cases we try to fit an instruction in 32 or 64 bits
Other decisions to make
How many registers?
0, 1 or infinite
32 or 64
if we use 64, we need 6 bits to specify registers as operands
Separate Integer and Floating point registers
64 of each kind?
implied by operations (Integer-Add means use integer registers)
Special purpose registers (PC, Status register, Stack-pointer (?))

What operations should we support?
Arithmetic – integer and floating point
Add, Subtract, Multiply and Divide
Logic (bit, byte or word)
AND, OR, NOT, …
Compare: EQ, NE, GE, LT,….
Conditional and unconditional branches
Delayed branch?

Delayed Branches
Even if the condition is satisfied, execute the next instruction before branching.
BEQ R1, R2, addr1
Add R3, R4, R5
……
addr1: Mult R3, R4, R5

Consider what happens if R1 equals R2
Normally, if R1=R2, the Add is not executed. But, in delayed branches, add will be executed whether the branch is taken or not.

What is the advantage?
The main advantage is in pipelines -- by the time branch is decoded, the next instruction is already fetched.

In delayed branches, we will not discard the next instruction even if branch is taken
Switch (i) {
    case '0': ...
    case '1': ...
    ....
}

Rs contains i; (R_I + displacement) contains the starting address of a table of addresses representing the branch addresses for each case clause.

---

**Function calls/ Subroutine calls.**

what are the necessary actions that must take place?

At call
- Save PC (return address)
- Pass parameters,
- Save registers  
  **By caller or by callee?**

At return
- Use the saved return address to return
- Restore registers -- again by caller or callee?

How about an instruction like **store multiple** registers and **load multiple**
to help with save/restore registers?

**What are the choices in implementation of function call?**
CSCE 4610: Computer Architecture

MIPS style: JAL – jump and link

PC is saved in R31 ($ra or return address register)
By convention you can pass some parameters through registers
R4-R7 ($a0-$a3 or arguments registers)
Results can also be returned through registers
R2-R3 ($v0-$v1)
Callee Saves registers R16-R23 ($s or save registers)
Caller Saves registers R8-R15 ($t or temporary registers)