Review:

- Execution time of a program
- Arithmetic Average, Weighted Arithmetic Average
- Geometric Mean
- Benchmarks, kernels and synthetic benchmarks
- Computing CPI for branch instructions
- CPI with cache misses
- Amdahl’s law
- MTTF and MTTR

HW #1  1.3, 1.5, 1.9, 1.12
Due: Sept 12, 2018

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

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>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>
</tr>
<tr>
<td>PUSH B</td>
<td>ADD B</td>
<td>ADD R3, R1, B</td>
<td>LOAD R2, B</td>
</tr>
<tr>
<td>ADD</td>
<td>STORE C</td>
<td>STORE C, R3</td>
<td>ADD R3, R1, R2</td>
</tr>
<tr>
<td>POP C</td>
<td></td>
<td>STORE C, R3</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

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>Load A</th>
<th>Load R1, A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add B</td>
<td>Load R2, B</td>
</tr>
<tr>
<td>Store Temp</td>
<td>Add R3, R1, R2</td>
</tr>
<tr>
<td>Load A</td>
<td>Load R4, D</td>
</tr>
<tr>
<td>Mul D</td>
<td>Multi R5, R1, R4</td>
</tr>
<tr>
<td>Add Temp</td>
<td>Add R6, R3, R5</td>
</tr>
<tr>
<td>Store C</td>
<td>Store C, R6</td>
</tr>
</tbody>
</table>
Figure A.4 Advantages and disadvantages of the three most common types of general-purpose register computers. The notation \((m, n)\) means \(m\) memory operands and \(n\) total operands. In general, computers with fewer alternatives simplify the compiler’s task because there are fewer decisions for the compiler to make (see Section A.8). Computers with a wide variety of flexible instruction formats reduce the number of bits required to encode the program.

The number of registers also affects the instruction size because you need \(\log_2\) (number of registers) for each register specifier in an instruction. Thus, doubling the number of registers takes three extra bits for a register-register architecture, or about 10% of a 32-bit instruction.

See page A6 for a discussion of Pros and cons on these different ISAs.

<table>
<thead>
<tr>
<th>Type</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register-register ((m, n))</td>
<td>Simple, fixed-length instruction encoding.</td>
<td>Higher instruction count than architectures with memory references in instructions. More instructions and lower instruction density lead to larger programs, which may have some instruction cache effects.</td>
</tr>
<tr>
<td>Register-memory ((1, 2))</td>
<td>Data can be accessed without a sequence load instruction first. Instruction format tends to be easy to encode and yields good density.</td>
<td>Operands are not equivalent because a source operand in a binary operation is destroyed. Encoding a register number and a memory address in each instruction may restrict the number of registers. Clocks per instruction vary by operand location.</td>
</tr>
<tr>
<td>Memory-memory ((2, 2)) or ((3, 3))</td>
<td>Most compact. Doesn’t waste registers for temporaries.</td>
<td>Large variation in instruction size, especially for three-operand instructions. In addition, large variation in work per instruction. Memory access can create memory bottleneck. (Not used today.)</td>
</tr>
</tbody>
</table>

Note in most systems, we specify an address for each byte. So a long integer (8 bytes) takes up 8 addresses, integer (32 bytes) uses 4 addresses.

Another issue to consider is the difference between Big-endian and Little-endian.

Address modes

How do we locate data in memory?

So far we assumed that we needed 64 bits to specify a memory address. Can we specify all the 64 bits in the instruction?

We can reduce the number of bits needed in an instruction with different address modes.

But before talking about address modes.

Let us first talk about how data is mapped to memory.

Consider the following structure.
struct foo {
    char a;
    int b;
    bool c;
    double d;
    short e;
    float f;
    double g;
    char * cptr;
    float *fptr;
    int x;
};

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>e</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

---

**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 5610: Computer Architecture

### Addressing mode

<table>
<thead>
<tr>
<th>Addressing mode</th>
<th>Example instruction</th>
<th>Meaning</th>
<th>When used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>Add R₄, R₃</td>
<td>Regs[R₄] ← Regs[R₃] + Regs[R₃]</td>
<td>When a value is in a register</td>
</tr>
<tr>
<td>Immediate</td>
<td>Add R₄, #3</td>
<td>Regs[R₄] ← Regs[R₄] + 3</td>
<td>For constants</td>
</tr>
<tr>
<td>Displacement</td>
<td>Add R₄, 100(R₅)</td>
<td>Regs[R₄] ← Regs[R₄] + Mem[100 Regs[R₅]]</td>
<td>Accessing local variables</td>
</tr>
<tr>
<td>Register indirect</td>
<td>Add R₄, (R₁)</td>
<td>Regs[R₄] ← Regs[R₄] + Mem[Reg[R₄]]</td>
<td>Accessing using a pointer or a computed address</td>
</tr>
<tr>
<td>Indirect</td>
<td>Add R₃, (R₁+R₂)</td>
<td>Regs[R₃] ← Regs[R₃] + Mem[Reg[R₃]+Reg[R₃]]</td>
<td>Sometimes useful in array addressing; R₁ = base of array; R₂ = index amount</td>
</tr>
<tr>
<td>Direct or absolute</td>
<td>Add R₁, (1001)</td>
<td>Regs[R₁] ← Regs[R₁] + Mem[1001]</td>
<td>Sometimes useful for accessing static data; address constant may need to be large</td>
</tr>
<tr>
<td>Memory indirect</td>
<td>Add R₁, #[R₃]</td>
<td>Regs[R₁] ← Regs[R₁] + Mem[Reg[R₃]]</td>
<td>If R₃ is the address of a pointer p, then mode yields “p”</td>
</tr>
<tr>
<td>Autodereference</td>
<td>Add R₁, -(R₂)</td>
<td>Regs[R₁] ← Regs[R₁] - Mem[Reg[R₂]]</td>
<td>Autodereference can also act as pop/push to implement a stack</td>
</tr>
<tr>
<td>Scaled</td>
<td>Add R₁, 100(R₂)[R₂]</td>
<td>Regs[R₁] ← Regs[R₁] + Mem[100 Regs[R₂]] + Regs[R₂]</td>
<td>Used to index arrays. May be applied to any indexed addressing mode in some computers</td>
</tr>
</tbody>
</table>

### Separate field for mode

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  Rᵣ, Rₛ, #value

**Load-Auto-inc** Rᵣ, disp (Rₛ)

We allow addresses with load/store (and branch instructions)

And we limit the address mode to Register + displacement only

- Load Rᵣ, disp (Rₛ)
- Store Rᵣ, disp (Rₛ)
This shows how frequently different address modes have been used for different programs compiled.

Note that the last 3 modes are more commonly used.

-- note we can make displacement = 0 (will be register indirect mode)
or use R0 so the register value is always zero (will be absolute address)
Load R0, 0 (R0)
Store R1, disp (R0)

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

See the figure on page A-11
It shows that you rarely need more than 16 bits of displacement for addresses
CSCE 5610: Computer Architecture

For immediate values
ADD R1, R2, #4

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_i)

How many bits for displacement in Jump?

CSCE 5610: 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
CSCE 5610: Computer Architecture

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?

CSCE 5610 Sept. 06, 2018

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': ....
    ....
}

What about cases like:
Switch (i) {
    case '0': ....
    case '5': ....
    case '13': ....
}

Rs contains i; (Ri + 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 5610: 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-$a3 or arguments registers)
Results can also be returned through registers
R2-R3 ($t0-$t1)
Callee Saves registers R16-R23 ($s or save registers)
Caller Saves registers R8-R15 ($t or temporary registers)

Using Stack (PDP 11)
Store PC on stack
Store all arguments on stack
Store the number of arguments on stack
Possibly save registers on stack

Is the stack in memory or a dedicated hardware?

CSCE 5610: Computer Architecture

Register Windows -- Sun SPARC