CSE471—Computer Design and Organization

Prof. Steven Burns *Autumn 1994*

Lecture Notes: Second Day

Chapter 2 — Performance

- There is a lot here but I won't use much of it.
- Please read it all again.
- Pay attention to the definition of MIPS and the pitfalls associated with benchmarking.

MIPS (Million Instructions per Second)

- For a given program P on machine A,

$$\begin{aligned}
\text{MIPS}_{P,A} &= \frac{\text{Instruction count}_{P,A}}{\text{Execution time}_{P,A} \times 10^6} \\
&= \frac{\text{Instruction count}_{P,A}}{\text{CPU clocks}_{P,A} \times \text{Cycle time}_A \times 10^6} \\
&= \frac{\text{Instruction count}_{P,A} \times \text{Clock rate}_A}{\text{Instruction count}_{P,A} \times \text{CPI}_{P,A} \times 10^6} \\
&= \frac{\text{Clock rate}_A}{\text{CPI}_{P,A} \times 10^6}
\end{aligned}$$

- We also have:

$$\text{Execution time}_{P,A} = \frac{\text{Instruction count}_{P,A}}{\text{MIPS}_{P,A} \times 10^6}$$

Problems with MIPS

- MIPS does not depend on the instruction set. You cannot use it to compare machines with different ISAs.

- MIPS varies between programs on the same computer (a machine does not have a single MIPS rating.)
- MIPS may even vary inversely with performance. (Consider two compilers, C and D for the same machine. If C generates code with more instructions than D but D still runs more quickly, then it might be the case that C's code has a higher MIPS rating than D's.)
- *Peak* MIPS

Amdahl's Law

- *Pitfall: Expecting the improvement of one aspect of a machine to increase performance by an amount proportional to the size of the improvement.*
- Say your machine has two instruction types. On a typical program $\frac{4}{5}$ of the time is spent executing type A instructions; $\frac{1}{5}$ on type B.
- If we make your machine 4 times faster for type A instructions, how much faster is it overall?

$$\frac{4}{5 \times 4}T + \frac{1}{5}T \quad = \quad \frac{2}{5}T$$

- Our speedup is $\frac{5}{2}$ instead of 4.

Chapter 3—Assembly Language and Machine Code

- We're going to go fast, but you've seen this before in 378.
- ISA design decisions are all pragmatic. You need only one instruction to perform any computation. (See Problems 3.37, 3.38)
- The book uses the instruction set architecture of the MIPS R2000.
  - Load–Store machine, three operand arithmetic
  - 32-bit instruction and data words
  - 32 general purpose registers; named \$0, \$1, ..., \$31

Example of Register Instructions

- Consider the fragment of C code:

```
f = (g + h) - (i + j);
```

- A compiler (or human) might translate this code to the instruction sequence:

```
                    # reg 17 contains g
                    # reg 18 contains h
                    # reg 19 contains i
                    # reg 20 contains j
add  $8, $17, $18   # reg 8 contains g+h
add  $9, $19, $20   # reg 9 contains i+j
sub $16,  $8,  $9   # f, reg 16, gets (g+h)-(i+j)
```

Example of Load–Store Instructions

- Assume `A` is an array of 100 words (400 bytes). Let the first element of `A` be at memory location 2000.

- Variables `h` and `i` are in registers \$18 and \$19. Register \$20 contains the constant 4.

- The C code:

```
A[i] = h + A[i]
```

translates (because of byte addressing) to

```
mul  $9, $19, $20       # register 9 contains i*4
lw   $8, 2000($9)       # register 8 contains A[i]
add  $8, $18, $8        # register 8 contains h+A[i]
sw   $8, 2000($9)       # A[i] now contains h+A[i]
```

Machine Code

- Binary representation—numbers represented by high and low signals

- The MIPS instruction

```
add $8,$17,$18
```

is represented by the code (R-format):

0	17	18	8	0	32

which is really

000000	10001	10010	01000	00000	100000
op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

The I-Format

- There is another instruction layout, called the I-format, or immediate format.
- The MIPS instructions

```
lw   $8, 2000($9)
sw   $8, 2000($9)
```

are represented as:

35	9	8	2000
43	9	8	2000

In binary, this is

100011	01001	01000	00000111 11010000
101011	01001	01000	00000111 11010000
op	rs	rt	address
6 bits	5 bits	5 bits	16 bits

Branch Instructions

- Need instructions to change the control flow.
- The C code fragment

```
if (i != j) {
   f = g + h;
}
f = f - i;
```

translates into the assembly code:

```
    beq $19,$20,L1  # goto L1 if i equals j
    add $16,$17,$18 # f = g + h (possibly skipped)
L1: sub $16,$16,$19 # f = f - i (always executed)
```

- `L1` is a code label, the address of the subtract instruction. The assembler translates this into a number.

Other Types of Branches

- MIPS machines use `beq`, `bne`, `slt`, and the fact that \$0 always contains the value zero to implement all types of branches.
- The command `slt $8,$19,$20` sets register \$8 to 1 if the contents of \$19 is less than \$20; otherwise \$8 gets 0.
- For example, the pseudoinstruction

  ```
  ble $19, $20, L1
  ```

  is translated by the assembler into

  ```
  slt $1, $20, $19  # $1 gets 0 if $20 >= $19
  beq $1, $0, L1
  ```

  Register \$1 is reserved for use by the assembler.

Supporting Procedure Calls

- The *jump* instruction takes one argument; it jumps to that address. (J-type)
- The *jump-and-link* instruction takes one argument; it jumps to that address and simultaneously stores the address of the next instruction in register \$31. (J-Type)
- The *jump-register* instruction jumps (unconditionally) to the address in the register specified by its only argument. (R-Type)
- What's really going on is the PC (program counter or instruction address register) is being examined and modified.

Jump Instruction Format

- The MIPS instructions

```
jal  10000
j    10000
```

are represented as (byte addresses replaced by word addresses):

3	2500
2	2500

In binary, this is

000011	0000 00000000 00100111 000100
000010	0000 00000000 00100111 000100
op	word address
6 bits	26 bits

Software Conventions For Procedure Calls

- To implement nested procedure calls, the return addresses must be saved. A *stack* is typically used to implement this.
- Not only the return address, but also the parameters and local variables need to be saved.
- The compilers and operation systems supporting the MIPS architecture use register $29 as a stack pointer. (Stacks by convention grow downward.)
- Caller save for registers.
- Callee save for registers.

Other MIPS Addressing Modes

- We can translate the C statement

```
i = i + 17
```

into

```
addi $29,$29,17  # $29 = $29 + 17
```

instead of loading the constant 17 from memory and then adding.

- Such instructions contain *immediate* operands (the constant 17).
- Represented in I-format.
- Also `slti`, `lui`.

Addressing in Branches and Jumps

- Branch addresses (stored in the I-format) are 16-bits. These branch addresses are word addresses relative to the instruction after the branch. Thus if the branch is taken,

$$\text{PC} \leftarrow (\text{PC} + 4) + 4 * \text{branch address}$$

- Jump addresses (stored in the J-format) are 26 bits and absolute (not relative to the PC).

$$\text{PC}_{27..0} \leftarrow 4 * \text{jump address}$$

(The 4 highest-order bits of the PC remain unchanged.)

Guiding Principles

1. *Smaller is faster.* This is why there are only 32 registers.
2. *Simplicity favors regularity.* Single size instructions, register fields in same place.
3. *Good design demands compromise.* Only `bne` and `beq`. 16-bit address offsets.
4. *Make the common case fast.* PC-relative addressing, immediate instructions for constants.

Assembler

- WARNING! the assembler is hiding some of the underlying complexity:
  - Pseudoinstructions
  - Delayed branches
  - Delayed loads
  - Large immediate constants
  - Long distance branches