Instruction Set Architecture

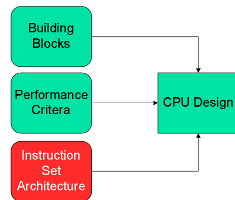
Chapter 2 – P & H

Instruction sets

- Language understood by the CPU
- Instruction sets regardless of CPU type tend to contain
  - Arithmetic instructions
    - Though some CPUs don't support certain types of arithmetic operation
  - Instructions for reading and writing to memory
  - Basic control instructions
- Their conventions can be quite different

Introduction

- Instruction set architecture interface between programmer and CPU
  - Good ISA makes program and CPU design easier
- P&H is based around MIPS architecture



WRAMP – Verses MIPS

- Addressing
  - MIPS – byte Addressable verses WRAMP - word addressable
  - MIPS can address 2^{32} locations verses 2^{19} on WRAMP
  - Words have to be aligned to word boundaries on MIPS
- General purpose registers
  - MIPS – 32 verses WRAMP at 8
  - On MIPS can reference byte, half word or word

WRAMP vs. MIPS

- Subroutine conventions
  - MIPS uses combination of stack and registers to pass parameters vs. stack only on WRAMP
  - $31 is returns address on MIPS verses $7 on WRAMP
  - Mixture of caller and callee saved registers on MIPS
- ISA not quite as regular on MIPS

CPU Design Principles

- Simplicity favours regularity
- Smaller is faster
- Good design demands good compromises
- Make the common case fast

MIPS arithmetic

- All instructions have 3 operands
- Operand order is fixed (destination first)

Example:

C code: `A = B + C`

MIPS code: `add $s0, $s1, $s2`

(associated with variables by compiler)

MIPS arithmetic

- Design Principle: simplicity favours regularity. Why?
- Of course this complicates some things...

C code:

```
A = B + C + D;
E = F - A;
```

MIPS code:

```
add $t0, $s1, $s2
add $s0, $t0, $s3
sub $s4, $s5, $s0
```

- Operands must be registers, only 32 registers provided
- Design Principle: smaller is faster. Why?

Registers vs. Memory

- On MIPS, arithmetic instructions operands must be registers
  — only 32 registers provided
- Compiler associates variables with registers
- What about programs with lots of variables?

Control	Memory	Input
Datapath		Output
Processor		I/O

Instructions

- Load and store instructions
- Example:

C code: `A[8] = h + A[8];`

MIPS code:

```
lw $t0, 32($s3)
add $t0, $s2, $t0
sw $t0, 32($s3)
```

- Store word has destination last
- Remember arithmetic operands are registers, not memory!

Our First Example

- Can we figure out the code?

```
swap(int v[], int k);
{ int temp;
    temp = v[k]
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

```
swap:
    muli $2, $5, 4
    add $2, $4, $2
    lw $15, 0($2)
    lw $16, 4($2)
    sw $16, 0($2)
    sw $15, 4($2)
    jr $31
```

Machine Language

- Instructions, like registers and words of data, are also 32 bits long
  - Example: `add $t0, $s1, $s2`
  - registers have numbers, `$t0=9, $s1=17, $s2=18`
- Instruction Format:

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

- *Can you guess what the field names stand for?*

Machine Language

- Consider the load-word and store-word instructions,
  - What would the regularity principle have us do?
  - New principle: Good design demands good compromises
- Introduce a new type of instruction format
  - I-type for data transfer instructions
  - other format was R-type for register
- Example: `lw $t0, 32($s2)`

35	18	9	32
op	rs	rt	16 bit number

- Where's the compromise?

Control

- Decision making instructions
  - alter the control flow,
  - i.e., change the "next" instruction to be executed
- MIPS conditional branch instructions:

```
bne $t0, $t1, Label
beq $t0, $t1, Label
```

- Example: if (i==j) h = i + j;

```
        bne $s0, $s1, Label
        add $s3, $s0, $s1
Label:      ....
```

Control

- MIPS unconditional branch instructions:
  `j label`
- Example:

```
if (i!=j)          beq $s4, $s5, Lab1
    h=i+j;         add $s3, $s4, $s5
else               j Lab2
    h=i-j;         Lab1: sub $s3, $s4, $s5
                   Lab2: ...
```

- *Can you build a simple for loop?*

Control Flow

- We have: beq, bne, what about Branch-if-less-than?
- New instruction:

```
                         if  $s1 < $s2 then
                             $t0 = 1
slt $t0, $s1, $s2        else
                             $t0 = 0
```

- Can use this instruction to build "`blt $s1, $s2, Label`"
  — can now build general control structures
- Note that the assembler needs a register to do this,
  — there are policy of use conventions for registers

2