Computer Architecture

Instructions

- Language of the Machine
- More primitive than higher level languages
  e.g., no sophisticated control flow

- Very restrictive, e.g., M IPS Arithmetic Instructions

- We'll be working with the MIPS instruction set architecture
  similar to other architectures developed since the 1980's
  used by NEC, Nintendo, Silicon Graphics, Sony

- Design goals: maximize performance and minimize cost,
  reduce design time

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 favors regularity. Why?
Of course this complicates some things...

C code:

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

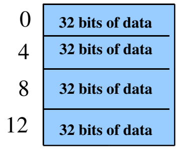
MIPS code:

```
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?

Memory Organization

Bytes are nice, but most data items use larger "words"
For MIPS, a word is 32 bits or 4 bytes.

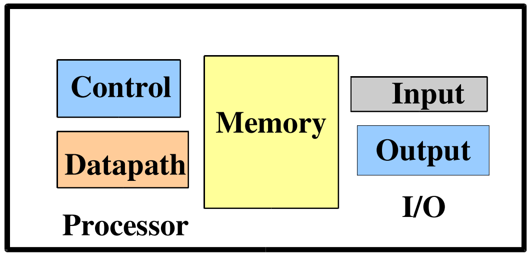


Register holds 32 bits

232 bytes with byte addresses from 0 to 232–1
230 words with byte addresses 0, 4, 8, ... 232–4
Words are aligned i.e., what are the least 2 significant bits of a word addre

Registers vz Memory

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



Load and Store Instructions

Load and store instructions
Example:

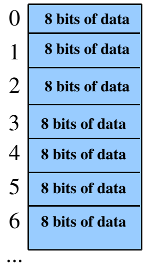
C code:
```
A[8] = h + A[8];
```

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!

Memory Organization

Viewed as a large, single–dimension array, with an address.
A memory address is an index into the array
<u>Byte addressing</u> means: index points to a byte of memory.



Machine Language

- Instructions, like registers and words of data, are 32 bits
  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 a compromise
- 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?

Example

```
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
```

So far we have learned

MIPS

—loading words but addressing bytes
—arithmetic on registers only

Instruction	RTL Notation
`add $s1, $s2, $s3`	`$s1 = $s2 + $s3`
`sub $s1, $s2, $s3`	`$s1 = $s2 – $s3`
`lw $s1, 100($s2)`	`$s1 = Memory[$s2+100]`
`sw $s1, 100($s2)`	`Memory[$s2+100] = $s1`

Control

- MIPS unconditional branch instructions:

```
j  label
```

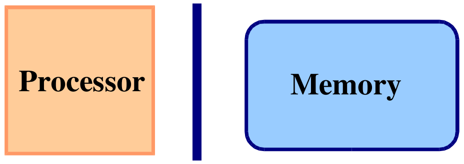
- <u>Example:</u>

```
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` ?

Stored Program Concept

Instructions are bits
Programs are stored in memory
—to be read or written just like data



Memory for data, programs, compilers, etc.

Fetch & Execute Cycle
- Instructions are fetched and put into a special register
- Bits in the register "control" the subsequent actions
- Fetch the "next" instruction and continue

Review so far

<u>Instruction</u>	<u>RTL Notation</u>
`add $s1,$s2,$s3`	\$s1 = \$s2 + \$s3
`sub $s1,$s2,$s3`	\$s1 = \$s2 – \$s3
`lw $s1,100($s2`	\$s1 = Memory[\$s2+100]
`sw $s1,100($s2)`	`Memory[$s2+100] = $s1`
`bne $s4,$s5,L`	Next instr. is at Label if \$s4 ° \$s5
`beq $s4,$s5,L`	Next instr. is at Label if \$s4 = \$s5
`j Label`	Next instr. is at Label

<u>Formats</u>:

Op	rs	rt	rd	shamt	fun

Op	rs	rt	16 bit address

Op	26 bit address

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: ....
```

Larger Constants

We'd like to be able to load a 32 bit constant into a register
Must use two instructions, new "load upper immediate" instruction

```
lui  $t0, 1010101010101010
```

1010101010101010	0000000000000000

Then must get the lower order bits right, i.e.,

```
ori  $t0, $t0, 1010101010101010
```

1010101010101010	0000000000000000
0000000000000000	1010101010101010
1010101010101010	1010101010101010

Assembly vs Machine Language

- Assembly provides convenient symbolic representation
  - much easier than writing down numbers
  - e.g., destination first
- Machine language is the underlying reality
  - e.g., destination is no longer first
- Assembly can provide 'pseudoinstructions'
  - e.g., "`move $t0, $t1`" exists only in Assembly
  - would be implemented using "`add $t0,$t1,$zero`"
- When considering performance you should count real instructions

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

Constants

- Small constants are used quite frequently (50% of operands)
  e.g., A = A + 5;
  B = B + 1;
  C = C – 18;
- Solutions? Why not?
  - put 'typical constants' in memory and load them.
  - create hard-wired registers (like $zero) for constants like one.
- MIPS Instructions:

```
addi $29, $29, 4
slti $8, $18, 10
andi $29, $29, 6
ori $29, $29, 4
```

How do we make this work?

Addresses in Branches and Jumps

- Instructions:

`bne $t4,$t5,Label`	Next instruction is at Label if `$t4 ° $t5`
`beq $t4,$t5,Label`	Next instruction is at Label if `$t4 = $t5`
`j Label`	Next instruction is at Label

- Formats:

Op	rs	rt	rd	sh	fun
Op	rs	rt	16 bit address		

- Addresses are not 32 bits
  —How do we handle this with load and store instructions?

Other Issues

- Things we are not going to cover

  support for procedures
  linkers, loaders, memory layout
  stacks, frames, recursion
  manipulating strings and pointers
  interrupts and exceptions
  system calls and conventions

- Some of these we'll talk about later

- We've focused on architectural issues
  – basics of MIPS assembly language and machine code
  – we'll build a processor to execute these instructions.

Addresses in Branches

- <u>Instructions</u>:

`bne $t4,$t5,Label`	Next instruction is at Label if `$t4°$t5`
`beq $t4,$t5,Label`	Next instruction is at Label if `$t4=$t5`

- <u>Formats</u>:

I	Op	rs	rt	16 bit address

- Could specify a register (like lw and sw) and add it to address
  – use Instruction Address Register (PC = program counter)
  – most branches are local (principle of locality)
- Jump instructions just use high order bits of PC
  – address boundaries of 256 MB

Overview of MIPS

- simple instructions all 32 bits wide
- very structured, no unnecessary baggage
- only three instruction formats

R	Op	rs	rt	rd	sh	fun
I	Op	rs	rt	16 bit address		
J	Op	26 bit address				

- rely on compiler to achieve performance
  —what are the compiler's goals?

- help compiler where we can

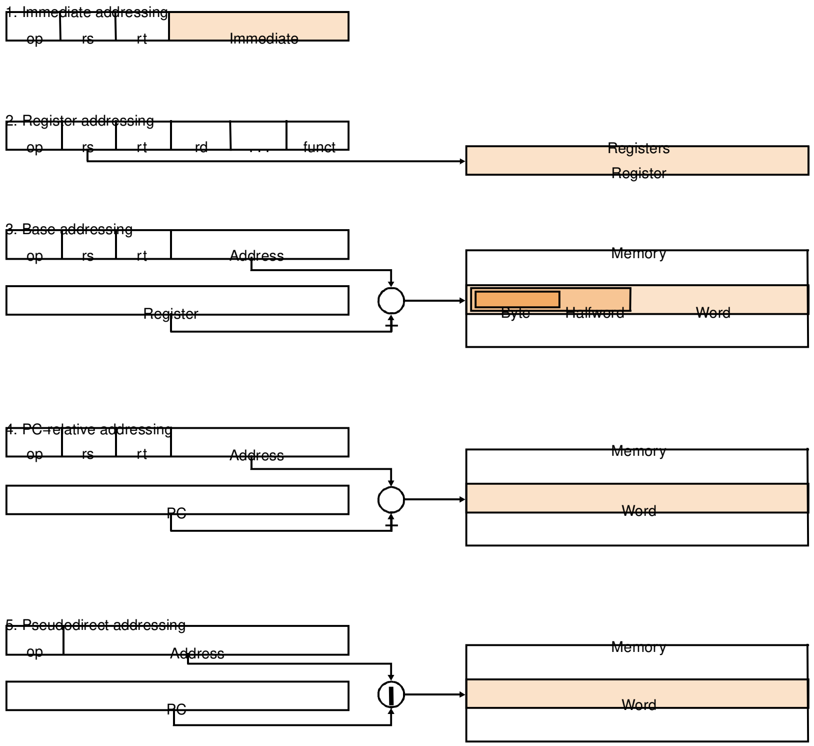
To Summarize

MIPS operands

Name	Example	Comments
32 registers	`$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register `$zero` always equals 0. Register `$at` is reserved for the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], ..., Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1, $s2, $s3`	`$s1 = $s2 + $s3`	Three operands; data in registers
	subtract	`sub $s1, $s2, $s3`	`$s1 = $s2 - $s3`	Three operands; data in registers
	add immediate	`addi $s1, $s2, 100`	`$s1 = $s2 + 100`	Used to add constants
Data transfer	load word	`lw $s1, 100($s2)`	`$s1 = Memory[$s2 + 100]`	Word from memory to register
	store word	`sw $s1, 100($s2)`	`Memory[$s2 + 100] = $s1`	Word from register to memory
	load byte	`lb $s1, 100($s2)`	`$s1 = Memory[$s2 + 100]`	Byte from memory to register
	store byte	`sb $s1, 100($s2)`	`Memory[$s2 + 100] = $s1`	Byte from register to memory
	load upper immediate	`lui $s1, 100`	`$s1` = 100 * 2^{16}	Loads constant in upper 16 bits
Conditional branch	branch on equal	`beq $s1, $s2, 25`	if (`$s1 == $s2`) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1, $s2, 25`	if (`$s1 != $s2`) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1, $s2, $s3`	if (`$s2 < $s3`) `$s1` = 1; else `$s1` = 0	Compare less than; for beq, bne
	set less than immediate	`slti $s1, $s2, 100`	if (`$s2 < 100`) `$s1` = 1; else `$s1` = 0	Compare less than constant
Uncondi– tional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to `$ra`	For switch, procedure return
	jump and link	`jal 2500`	`$ra` = PC + 4; go to 10000	For procedure call



Alternative Architectures

- Design alternative:
  – provide more powerful operations
  – goal is to reduce number of instructions executed
  – danger is a slower cycle time and/or a higher CPI
- Sometimes referred to as "RISC vs. CISC"
  – virtually all new instruction sets since 1982 have been RISC
  – VAX: minimize code size, make assembly language easy
    instructions from 1 to 54 bytes long!
- We'll look at PowerPC and 80x86

PowerPC

- Indexed addressing
  – example: `lw $t1,$a0+$s3   #$t1=Memory[$a0+$s3]`
  – What do we have to do in MIPS?
- Update addressing
  – update a register as part of load (for marching through arrays)
  – example:
    `lwu $t0,4($s3)  #$t0=Memory[$s3+4];$s3=$s3+4`
  – What do we have to do in MIPS?
- Others:
  – load multiple/store multiple
  – a special counter register "bc Loop"
    decrement counter, if not 0 goto loop

Summary

- Instruction complexity is only one variable
  – lower instruction count vs. higher CPI / lower clock rate

- Design Principles:
  – simplicity favors regularity
  – smaller is faster
  – good design demands compromise
  – make the common case fast

- Instruction set architecture
  – a very important abstraction indeed!

80x86

1978: The Intel 8086 is announced (16 bit architecture)

1980: The 8087 floating point coprocessor is added

1982: The 80286 increases address space to 24 bits, +instructions

1985: The 80386 extends to 32 bits, new addressing modes

1989–1995: The 80486, Pentium, Pentium Pro add a few instructions
(mostly designed for higher performance)

1997: MMX is added

80x86: Dominant Architecture

- Complexity:
  – Instructions from 1 to 17 bytes long
  – one operand must act as both a source and destination
  – one operand can come from memory
  – complex addressing modes
    e.g., "base or scaled index with 8 or 32 bit displacement"

- Saving grace:
  – the most frequently used instructions not too difficult to build
  – compilers avoid the portions of the architecture that are slow

"w*hat the 80x86 lacks in style is made up in quantity, making it beautiful from the right perspective*"