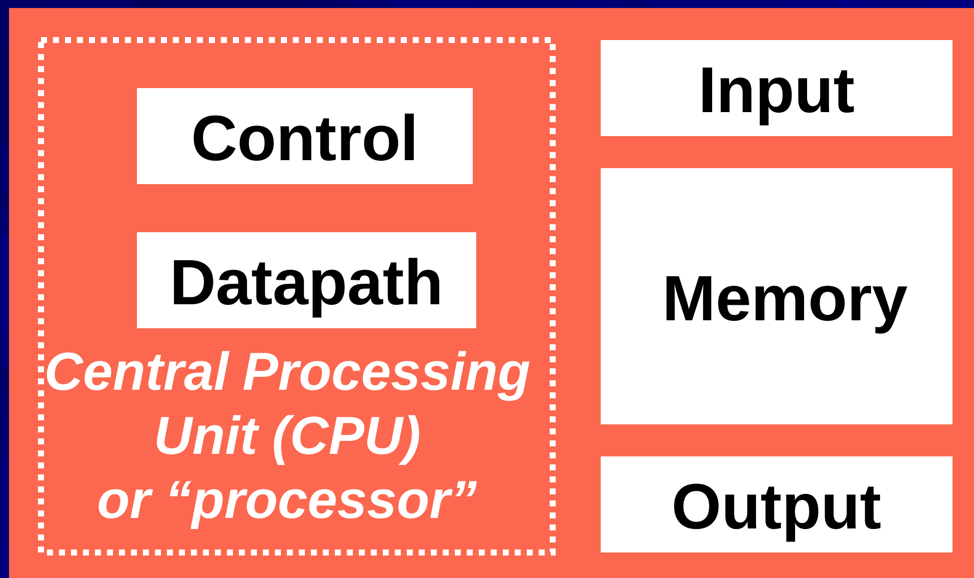
ELEC 5200-001/6200-001 Computer Architecture and Design Fall 2013

Instruction Set Architecture (Chapter 2)

Victor P. Nelson, Professor & Asst. Chair
Vishwani D. Agrawal, James J. Danaher Professor
Department of Electrical and Computer Engineering
Auburn University, Auburn, AL 36849

Designing a Computer



FIVE PIECES OF HARDWARE

Start by Defining ISA

- What is instruction set architecture (ISA)?
- ISA
  - Defines registers
  - Defines data transfer modes (instructions) between registers, memory and I/O
  - There should be *sufficient* instructions to efficiently translate any program for machine processing
- Next, define instruction set format – binary representation used by the hardware
  - Variable-length vs. fixed-length instructions

Types of ISA

- Complex instruction set computer (CISC)
  - Many instructions (several hundreds)
  - An instruction takes many cycles to execute
  - Example: Intel Pentium
- Reduced instruction set computer (RISC)
  - Small set of instructions (typically 32)
  - Simple instructions, each executes in one clock cycle – ***REALLY? Well, almost.***
  - Effective use of pipelining
  - Example: ARM

On Two Types of ISA

- Brad Smith, “ARM and Intel Battle over the Mobile Chip’s Future,” Computer, vol. 41, no. 5, pp. 15-18, May 2008.
- Compare 3Ps:
  - Performance
  - Power consumption
  - Price

Pipelining of RISC Instructions

Fetch Instruction → **Decode Opcode** → **Fetch Operands** → **Execute Operation** → **Store Result**

Although an instruction takes five clock cycles, one instruction can be completed every cycle.

Growth of Processors

- **Language of the Machine**
- **We'll be working with the MIPS instruction set architecture**
  - **similar to other architectures developed since the 1980's**
  - **Almost 100 million MIPS processors manufactured in 2002**
  - **used by NEC, Nintendo, Cisco, Silicon Graphics, Sony, …**

MIPS Instruction Set (RISC)

- Instructions execute simple functions.
- Maintain regularity of format – each instruction is one word, contains *opcode* and *arguments*.
- Minimize memory accesses – whenever possible use registers as arguments.
- Three types of instructions:
  - Register (R)-type – only registers as arguments.
  - Immediate (I)-type – arguments are registers and numbers (constants or memory addresses).
  - Jump (J)-type – argument is an address.

MIPS Arithmetic Instructions

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

Example:

C code: `a = b + c;`

MIPS 'code': `add a, b, c`

"The natural number of operands for an operation like addition is three… requiring every instruction to have exactly three operands conforms to the philosophy of keeping the hardware simple"

Arithmetic Instr. (Continued)

- Design Principle: simplicity favors regularity.
- Of course this complicates some things...

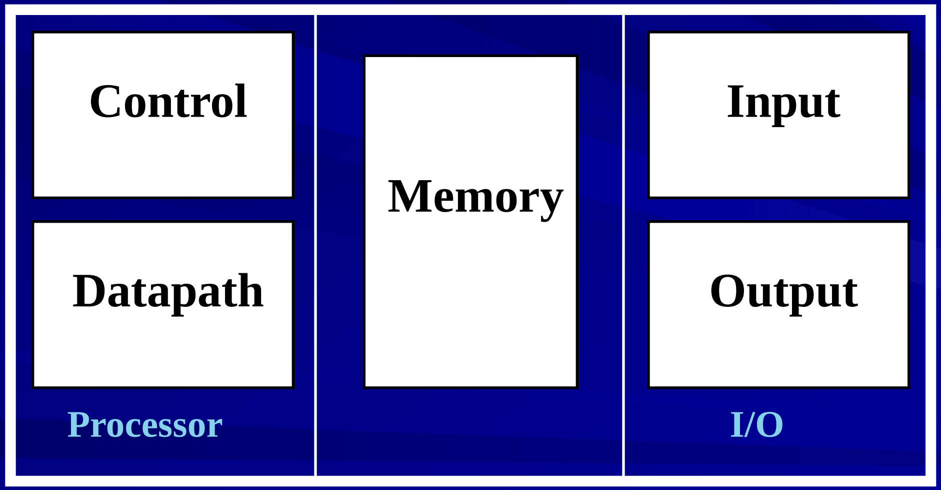
C code: `a = b + c + d;`

MIPS code:
```
add a, b, c
add a, a, d
```

- Operands must be registers (why?) *Remember von Neumann bottleneck.*
- 32 registers provided
- Each register contains 32 bits

Registers vs. Memory

- Arithmetic instructions operands must be registers
  - **32 registers provided**
- Compiler associates variables with registers.
- What about programs with lots of variables? *Must use memory.*



Memory Organization

- Viewed as a large, single-dimension array, with an address.
- A memory address is an index into the array.
- "Byte addressing" means that the index points to a byte of memory.

	Byte 0	byte 1	byte 2	byte 3
32 bit word	8 bits of data	8 bits of data	8 bits of data	8 bits of data
32 bit word	8 bits of data	8 bits of data	8 bits of data	8 bits of data
32 bit word	8 bits of data	8 bits of data	8 bits of data	8 bits of data
32 bit word	8 bits of data	8 bits of data	8 bits of data	8 bits of data
.	8 bits of data	8 bits of data	8 bits of data	8 bits of data

.

.

byte 4

byte 10

"Little endian" vs "Big endian"

Store 32-bit number
0x347A4302

LSB at lowest address	
02	100
43	101
7A	102
34	103

103	102	101	100
34	7A	43	02

LSB at highest address	
34	100
7A	101
43	102
02	103

100	101	102	103
34	7A	43	02

Memory Organization

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

word addresses

0	32 bits of data
4	32 bits of data
8	32 bits of data
12	32 bits of data
.	32 bits of data

...

Registers hold 32 bits of data

Use 32 bit address

- **2^{32} bytes with addresses from 0 to $2^{32} - 1$**
- **2^{30} words with addresses 0, 4, 8, ... $2^{32} - 4$**
- **Words are aligned**
  i.e., what are the least 2 significant bits of a word address?

Instructions

- Load and store instructions
- Example:

C code: `A[12] = h + A[8];`

MIPS code:

```
lw $t0, 32($s3)   #addr of A in reg s3
add $t0, $s2, $t0 #h in reg s2
sw $t0, 48($s3)
```

- Can refer to registers by name (e.g., $s2, $t2) or number ($18, $10)
- Store word has destination last
- Remember arithmetic operands are registers, not memory!

Can't write: `add 48($s3), $s2, 32($s3)`

Policy of Register Usage (Conventions)

Name	Register number	Usage
`$zero`	0	the constant value 0
`$v0-$v1`	2-3	values for results and expression evaluation
`$a0-$a3`	4-7	arguments
`$t0-$t7`	8-15	temporaries
`$s0-$s7`	16-23	saved
`$t8-$t9`	24-25	more temporaries
`$gp`	28	global pointer
`$sp`	29	stack pointer
`$fp`	30	frame pointer
`$ra`	31	return address

Register 1 ($at) reserved for assembler, 26-27 for operating system

Our First Example

- Can we figure out the code of subroutine?

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

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

- Initially, k is in reg 5; base address of v is in reg 4; return addr is in reg 31

What Happens?

.
.
call swap
. ← ***return address***
.
.

- When the program reaches “call swap” statement:
  - Jump to swap routine
    - Registers 4 and 5 contain the arguments (register convention)
    - Register 31 contains the return address (register convention)
  - Swap two words in memory
  - Jump back to return address to continue rest of the program

Memory and Registers

byte addr.	Memory
0	Word 0
4	Word 1
8	Word 2
12	
.	
4n	v[0] (Word n)
.	v[1] (Word n+1)
.	
.	
4n+4k	v[k] (Word n+k)
.	v[k+1] (Word n+k+1)

Register	
Register 0	
Register 1	
Register 2	
Register 3	
Register 4	4n
Register 5	k
.	
.	
Register 31	*Ret. addr.*

Our First Example

- Now figure out the code:

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

→

```
swap:
   sll $2, $5, 2
   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've Learned

- MIPS
  - — loading words but addressing bytes
  - — arithmetic on registers only
- Instruction / Meaning

Instruction	Meaning
`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`

Machine Language

- Instructions, like registers and words of data, are also 32 bits long
  - Example: **`add $t1, $s1, $s2`**
  - registers are numbered, **`$t1=8, $s1=17, $s2=18`**

- Instruction Format:

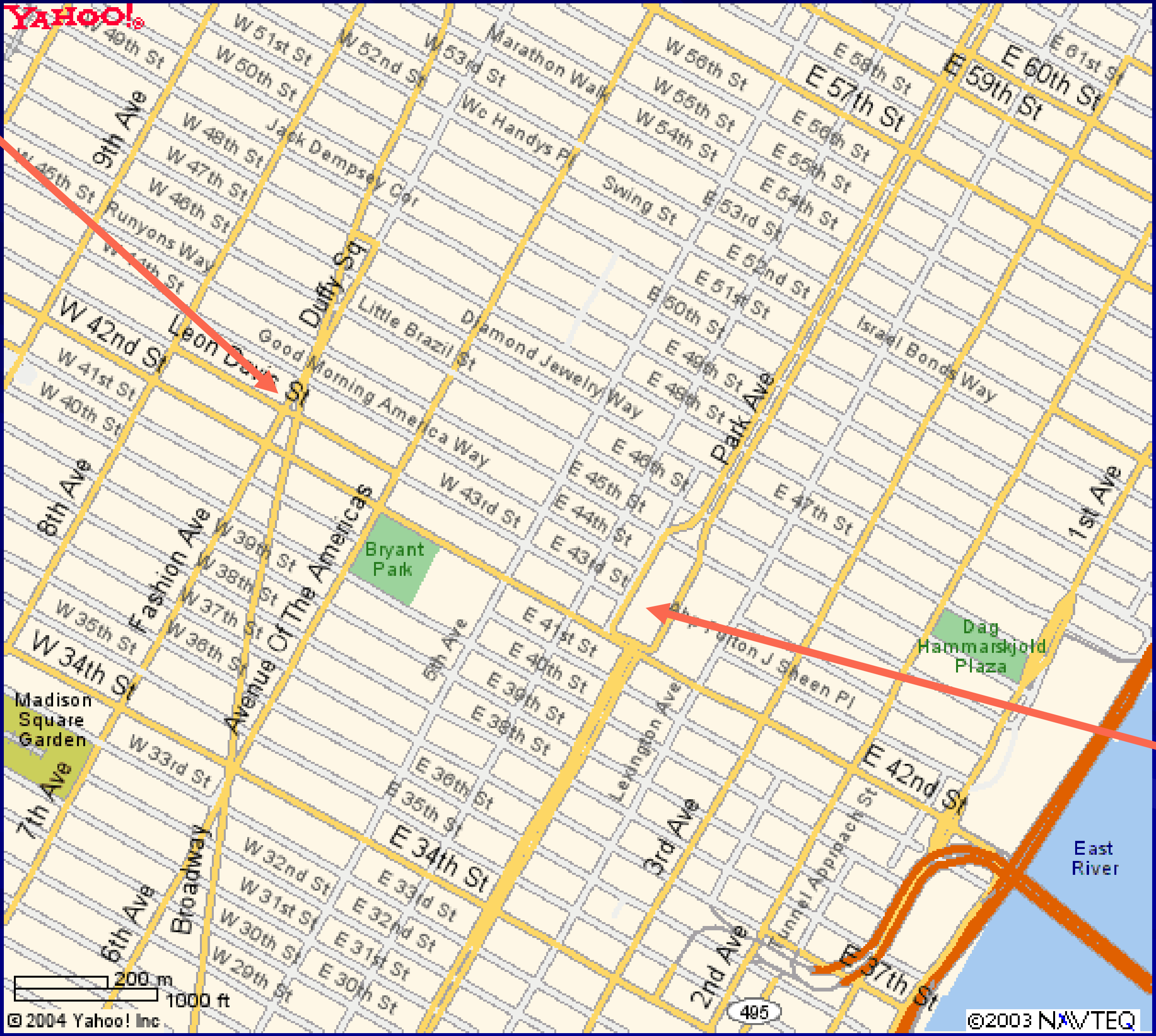
000000	10001	10010	01000	00000	100000

opcode	rs	rt	rd	shamt	funct

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

Violating Regularity for a Good Cause

Times Square



Grand Central Station

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)`

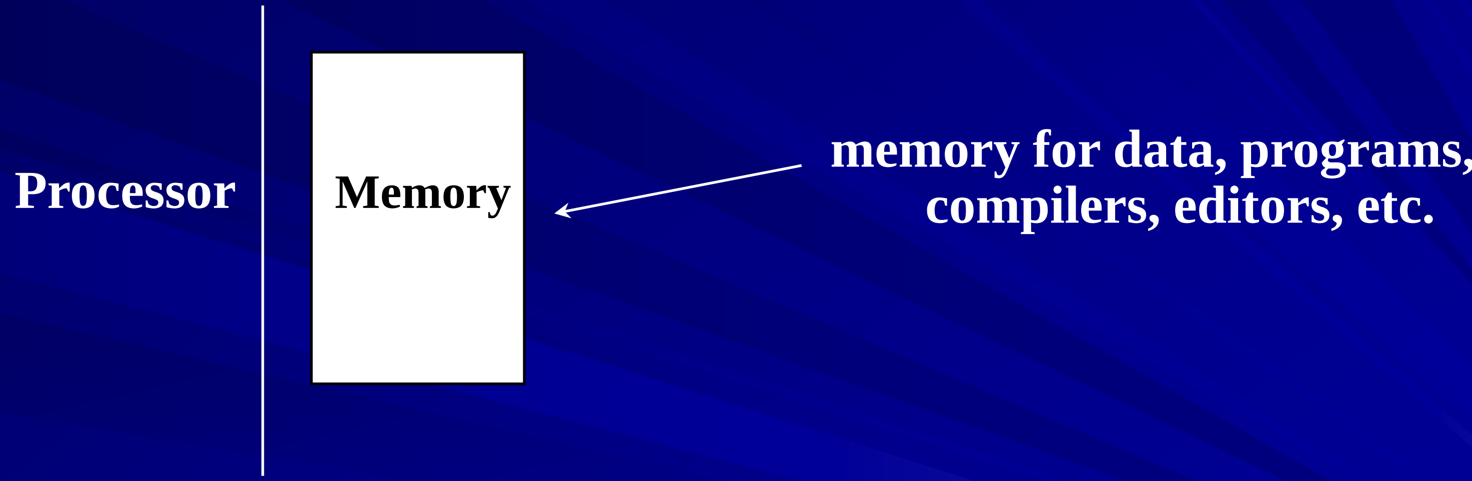
35	**18**	**9**	**32**

opcode	**rs**	**rt**	**16 bit number**

- Where's the compromise?

Stored Program Concept

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



- **Fetch and Execute Cycles**
  - **Instructions are fetched and put into a special register**
  - **Opcode bits in the register "control" the subsequent actions**
  - **Fetch the “next” instruction and continue**

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?

So Far We've Learned

- Instruction / Meaning

Instruction	Meaning
`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,Label`	`Next instr. is at Label if $s4 ≠ $s5`
`beq $s4,$s5,Label`	`Next instr. is at Label if $s4 = $s5`
`j Label`	`Next instr. is at Label`

- Formats:

R	op	rs	rt	rd	shamt	funct
I	op	rs	rt	16 bit address		
J	op	26 bit address				

Three Ways to Jump: j, jr, jal

- j *instr* # jump to machine instruction *instr* (unconditional jump)
- jr $ra # jump to address in register ra (used by callee to go back to caller)
- jal *addr* # set $ra = PC+4 and go to *addr* (jump and link; used to jump to a procedure)

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 new "pseudoinstruction"

```
blt $s1, $s2, Label
```

- Note that the assembler needs a register to do this,
  — there are policy of use conventions for registers

Pseudoinstructions

- blt $s1, $s2, reladdr
- Assembler converts to:

```
slt $1, $s1, $s2
bne $1, $zero, reladdr
```

- Other pseudoinstructions: bgt, ble, bge, li, move
- Not implemented in hardware
- Assembler expands pseudoinstructions into machine instructions
- Register 1, called $at, is reserved for converting pseudoinstructions into machine code.

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
```

- Design Principle: Make the common case fast. *Which format?*

How About 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
```

filled with zeros

1010101010101010	0000000000000000

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

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

	1010101010101010	0000000000000000
ori	0000000000000000	1010101010101010
	1010101010101010	1010101010101010

Assembly Language 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
  - implemented using “add $t0, $t1, $zero”
- When considering performance you should count real instructions and clock cycles

Overview of MIPS

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

R	op	rs	rt	rd	shamt	funct
I	op	rs	rt	16 bit address		
J	op	26 bit address				

- rely on compiler to achieve performance

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:

I	op	rs	rt	16 bit rel. address
J	op	26 bit absolute address		

Addresses in Branches

- 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

- Formats:

-2^{15} to $2^{15}-1$ ~ ±32 Kwords

op	**rs**	**rt**	**16 bit address**

op	**26 bit address**

2^{26} = 64 Mwords

- Relative addressing
  – with respect to PC (program counter)
  – most branches are local (principle of locality)
- Jump instruction just uses high order bits of PC
  – address boundaries of 256 MBytes (maximum jump 64 Mwords)

Example: Loop in C (p. 74)

```
while  ( save[i] == k )
        i += 1;
```

Given a value for k, set i to the index of element in array save [ ] that does not equal k.

MIPS Code for While Loop

Compiler assigns variables to registers:

$s3 (reg 19) ← i initially 0
$s5 (reg 21) ← k
$s6 (reg 22) ← memory address where save [ ] begins

Then generates the following assembly code:

```
Loop: sll   $t1, $s3, 2       # Temp reg $t1 = 4 * i
      add   $t1, $t1, $s6     # $t1 = address of save[i]
      lw    $t0, 0($t1)       # Temp reg $t0 = save[i]
      bne   $t0, $s5, Exit    # go to Exit if save[i] ≠ k
      addi  $s3, $s3, 1       # i = i + 1
      j     Loop              # go to Loop
Exit:
```

Machine Code and Mem. Adresses

Memory Byte addr.	**Bits 31-26**	**25-21**	**20-16**	**15-11**	**10 – 6**	**5 – 0**	
80000	0	0	19	9	2	0	sll
80004	0	9	22	9	0	32	add
80008	35	9	8	0			lw
80012	5	8	21	Exit = +2			bne
80016	8	19	19	1			addi
80020	2	Loop = 20000 (memory word address)					j
80024	. . . . .						

Machine code

Note: $t0 ≡ Reg 8, $t1 ≡ Reg 9, $s3 ≡ Reg 19, $s5 ≡ Reg 21, $s6 ≡ Reg 22
temp temp i k save

Finding Branch Address *Exit*

- Exit = +2 is a 16 bit integer in bne instruction
  000101 01000 10101 0000000000000010 = 2
- $PC = 80016 is the byte address of the next instruction
  00000000000000010011100010010000 = 80016
- Multiply bne argument by 4 (convert to byte address)
  0000000000001000 = 8
- $PC ← $PC + 8
  00000000000000010011100010011000 = 80024
  Thus, *Exit* is memory byte address 80024.

Finding Jump Address *Loop*

- J 20000

  000010 00000000000100111000100000 = 20000

- $PC = 80024, when jump is being executed

  00000000000000010011100010011000 = 80024

- Multiply J argument by 4 (convert to byte address)

  0000000000010011100010000000 = 80000

- Insert four leading bits from $PC

  00000000000000010011100010000000 = 80000

  Thus, *Loop* is memory byte address 80000.

Summary: MIPS Registers and Memory

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.

Summary: MIPS Instructions

MIPS assembly language				
Category	**Instruction**	**Example**	**Meaning**	**Comments**
	add	`add $s1, $s2, $s3`	`$s1 = $s2 + $s3`	Three operands; data in registers
Arithmetic	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
	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
Data transfer	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
	branch on equal	`beq $s1, $s2, 25`	if (`$s1 == $s2`) go to PC + 4 + 100	Equal test; PC-relative branch
Conditional 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
	jump	`j 2500`	go to 10000	Jump to target address
Unconditional jump	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

Addressing Modes

Example

addi

add

lw, sw

beq, bne

j