7 September 2004

- Questions?
- Farmer's Days?
- Programming the Machine

Instructions:

- Language of the Machine
- More primitive than higher level languages
  e.g., no sophisticated control flow
- Very restrictive
  e.g., MIPS Arithmetic Instructions

- We'll be working with the MIPS instruction set architecture
  - similar to other architectures developed since the 1980's

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...

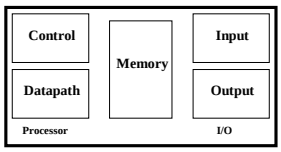
C code:
```
A = B + C + D;
E = F - A;
```

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?

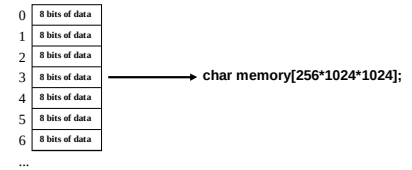
Registers vs. Memory

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



Memory Organization

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



Memory Organization

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

Registers hold 32 bits of data

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

Endians?

- What order are the bytes inside the word?
- Is byte 0 the high-order bits of word 0?
- Or is it the low order bits?

- "Big Endian" byte 0 is high-order bits [0 1 2 3]
  - Macintosh, SPARC
- "Little Endian" byte 0 is low-order bits [3 2 1 0]
  - Intel, DECStation

When would I care?

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!

So far we've learned:

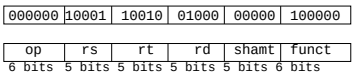
- MIPS
  — loading words but addressing bytes
  — arithmetic on registers only

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

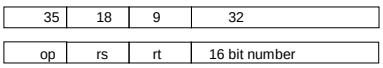
- Instructions, like registers and words of data, are also 32 bits long
  - Example: `add $t0, $s1, $s2`
  - registers have numbers, `$t0=8, $s1=17, $s2=18`

- Instruction Format:



Machine Language

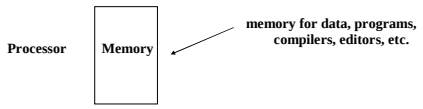
- Consider the load-word and store-word instructions,
  - What would the regularity principle have us do?
  - New principle: **Good design demands a compromise**
- Introduce a new type of instruction format
  - I-type for data transfer instructions
  - other format was R-type for register
- Example: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



- Where's the compromise?

Stored Program Concept

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



- 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

So far we've learned:

- MIPS
  — loading words but addressing bytes
  — arithmetic on registers only
- 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
```


Execution Example

Program Counter: 200

Memory(32 bits)

200	10001101000010010000000000000000	LW $9, 0($8)
204	00000001001001110100100000100000	ADD $9,$9,$7
208	10101101000010010000000000001000	SW $9, 8($8)
212		

Memory(32 bits)

112	8
116	13
120	21
124	34
128	55
132	89

Registers (32 bits)

6	1234
7	23
8	120
9	-314159
10	316

Instruction Register (32 bits)

	op	rs	rt	rd	shft	func
R	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

	op	rs	rt	offset
I	6 bits	5 bits	5 bits	16 bits

Execution Example: Fetch(200)

Program Counter: 200

Memory

200	10001101000010010000000000000000	LW $9, 0($8)
204	00000001001001110100100000100000	ADD $9,$9,$7
208	10101101000010010000000000001000	SW $9, 8($8)
212		

Memory

112	8
116	13
120	21
124	34
128	55
132	89

Registers

6	1234
7	23
8	120
9	-314159
10	316

Instruction Register

R						
I	100011	01000	01001	0000000000000000		
	35	8	9	0		

Execution Example: Execute(200)

Program Counter: 204

Memory

200	10001101000010010000000000000000	LW $9, 0($8)
204	00000001001001110100100000100000	ADD $9,$9,$7
208	10101101000010010000000000001000	SW $9, 8($8)
212		

Memory

112	8
116	13
120	21
124	34
128	55
132	89

Registers

6	1234
7	23
8	120
9	21
10	316

Instruction Register

R						
I	100011	01000	01001	0000000000000000		
	35	8	9	0		

Execution Example: Fetch(204)

Program Counter: 204

Memory

200	10001101000010010000000000000000	LW $9, 0($8)
204	00000001001001110100100000100000	ADD $9,$9,$7
208	10101101000010010000000000001000	SW $9, 8($8)
212		

Memory

112	8
116	13
120	21
124	34
128	55
132	89

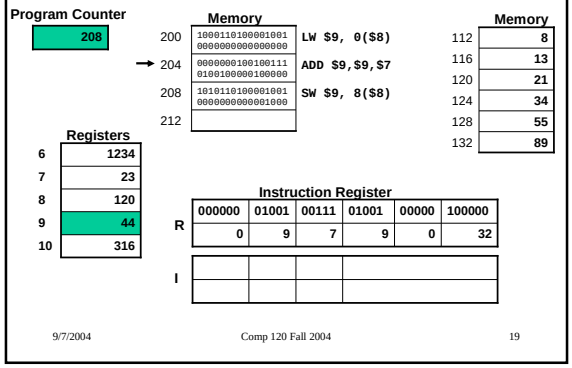
Registers

6	1234
7	23
8	120
9	21
10	316

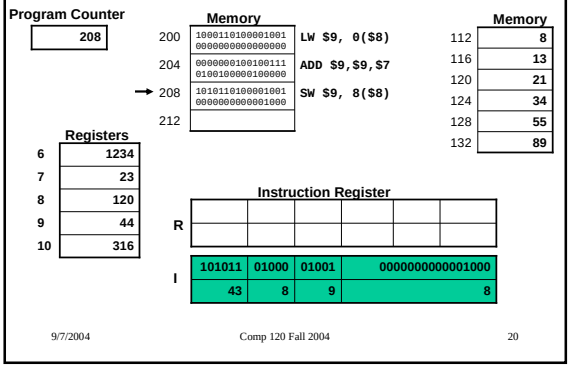
Instruction Register

R	000000	01001	00111	01001	00000	100000
	0	9	7	9	0	32
I						

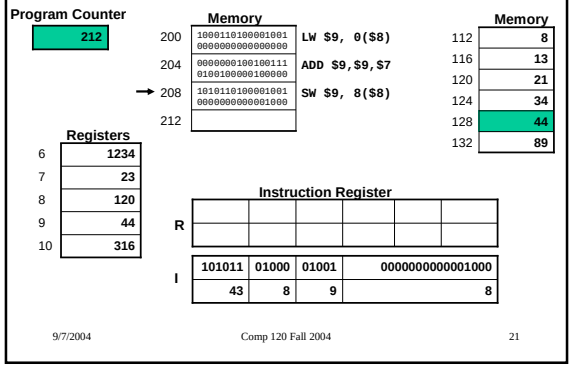
Execution Example: Execute(204)



Execution Example: Fetch(208)



Execution Example: Execute(208)



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: ...
```


So far:

- 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,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
```

- Formats:

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

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

Policy of Use Conventions

Name	Register number	Usage
`$zero`	0	the constant value 0
`$at`	1	reserved for the assembler
`$v0-$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