ECE 250 / CS 250
Computer Architecture

MIPS Instruction Set

Benjamin Lee

Slides based on those from Alvin Lebeck, Daniel Sorin, Andrew Hilton, Amir Roth, Gershon Kedem

Today's Lecture

Homework #1

- Due today @ 11:55pm
- Segmentation Faults
  - Could come from three places: LL node creation, LL sort, strings
  - LL node creation – start with lecture examples
  - LL traversal – create nodes with integers and sort them
  - Strings – look at strcpy and strcmp
- Late Policy – 10% penalty for 24 hours, 20% penalty for 48 hours

Outline

- MIPS: A specific ISA, we'll use it throughout semester
- Instructions and categories

Reading

- An overview of SPIM and the MIPS32 instruction set @ http://spimsimulator.sourceforge.net/HP_AppA.pdf
- MIPS Quick Ref (PDF in docs section of web site)

Review: Basic ISA Classes

Accumulator:

1 address	add A	acc ← acc + mem[A]
1+x address	addx A	acc ← acc + mem[A + x]

Stack:

0 address	add	tos ← tos + next (JAVA VM)

General Purpose Register:

2 address	add A B	A ← A + B
3 address	add A B C	A ← B + C

Load/Store:

3 address	add Ra Rb Rc	Ra ← Rb + Rc
	load Ra Rb	Ra ← mem[Rb]
	store Ra Rb	mem[Rb] ← Ra

Review: LOAD / STORE ISA

- Instruction set:

  add, sub, mult, div, ... only on operands in registers

  ld, st, to move data from and to memory, only way to access memory

Example: a*b - (a+c*b) (assume in memory)

	r1,	r2,	r3
ld r1, c	2,	?,	?
ld r2, b	2,	3,	?
mult r1, r1, r2	6,	3,	?
ld r3, a	6,	3,	4
add r1, r1, r3	10,	3,	4
mult r2, r2, r3	10,	12,	4
sub r3, r2, r1	10,	12,	2

7 instructions

a	4
b	3
c	2

Review: Using Registers to Access Memory

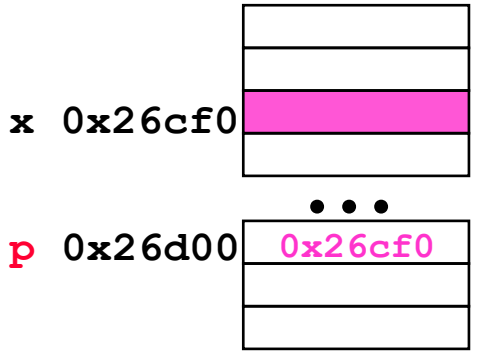
- Registers can hold memory addresses

Given

```
int x; int *p;
p = &x;
*p = *p + 8;
```

Instructions

```
ld r1, p              // r1 <- mem[p]
ld r2, r1             // r2 <- mem[r1]
add r2, r2, 0x8       // increment x by 8
st r1, r2             // mem[r1] <- r2
```

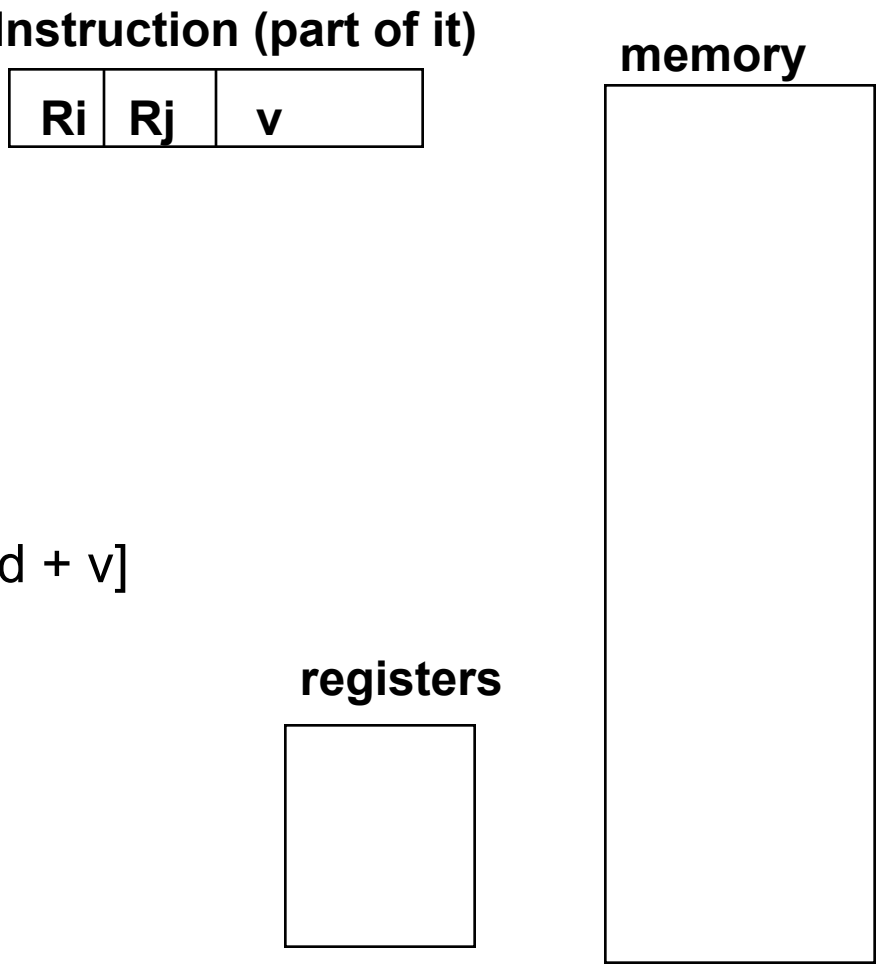


Operand Addressing Modes

- Register direct Ri
- Immediate (literal) v
- Direct (absolute) M[v]

- Register indirect M[Ri]
- Base+Displacement M[Ri + v]
- Base+Index M[Ri + Rj]
- Scaled Index M[Ri + Rj*d + v]
- Autoincrement M[Ri++]
- Autodecrement M[Ri - -]

- Memory Indirect M[M[Ri]]



Machine Readable Instructions

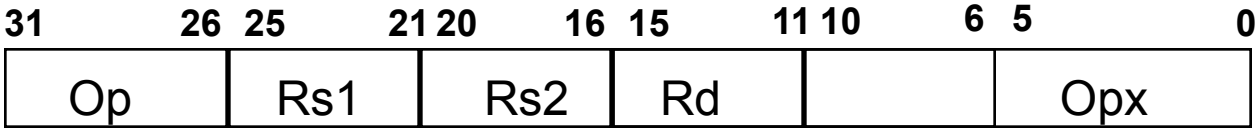
- Instructions are too abstract
  - add r1, r2, r3
- Need to specify instructions in machine readable form

- Instructions are bits with defined fields
  - E.g., floating point number has different fields

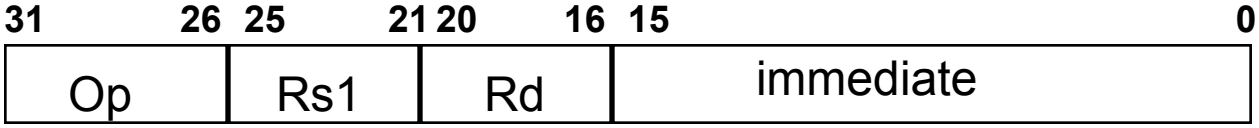
- Instruction format defines fields
  - Map instruction to binary values
  - Determine which bit positions correspond to which parts of the instruction (operation, operands, etc.)

Example: MIPS

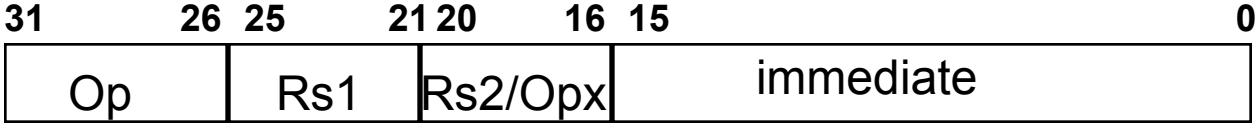
Register-Register



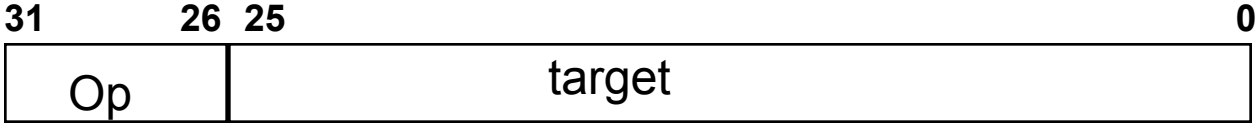
Register-Immediate



Branch



Jump / Call

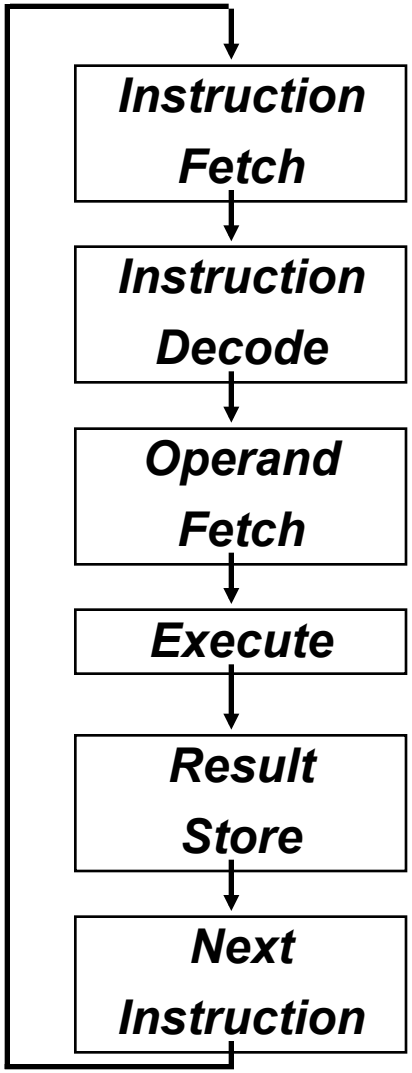


Stored Program Computer

- Instructions: a set of built-in operations
- Instructions and data are stored in the same memory
- Fetch-Execute cycle implemented in hardware

```
while (!done)
    fetch instruction
    execute instruction
```

Review: What Must ISA Specify?



- Instruction Format
  - what operation to perform?
- Location of operands and result
  - registers or memory?
  - how many explicit operands?
  - how are memory operands located?
- Data Type and Size
- Operations
- Successor instruction
  - jumps, conditions, branches

MIPS Instruction Categories

- Arithmetic
  - add, sub, mul, etc
- Logical
  - and, or, shift
- Data Transfer
  - load, store
  - MIPS is load/store architecture
- Conditional Branch
  - implement if, for, while… statements
- Unconditional Jump
  - support method invocation, function call, procedure calls

MIPS Instruction Set Architecture

- 3-Address Load/Store Architecture
- Register and immediate operand addressing for operations
- Immediate and displacement addressing for loads/stores
- 32-bit fixed size instruction format

- Examples (Assembly Language…details later):

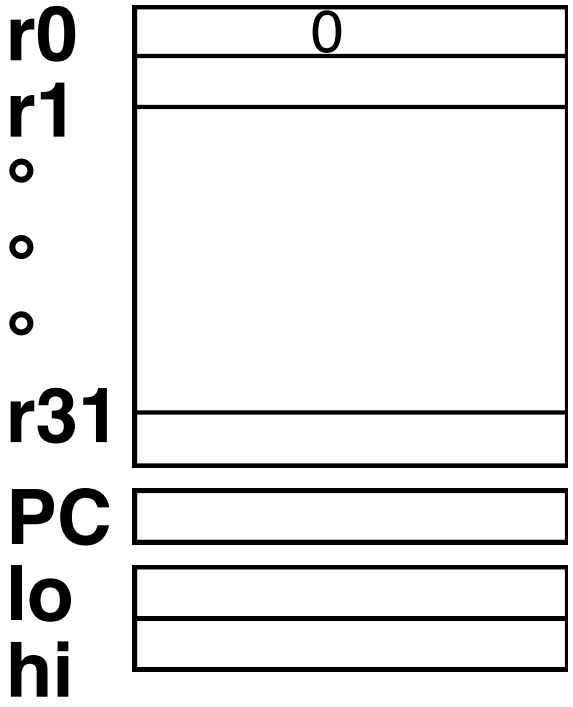
```
add     $1, $2, $3      #    $1 = $2 + $3
addi    $1, $1, 4       #    $1 = $1 + 4

lw      $1, 100 ($2)    #    $1 = Memory[$2 + 100]
sw      $1, 100 ($2)    #    Memory[$2 + 100] = $1

lui     $1, 100         #    $1 = 100 X 2^16
addi    $1, $3, 100     #    $1 =  $3 + 100
```

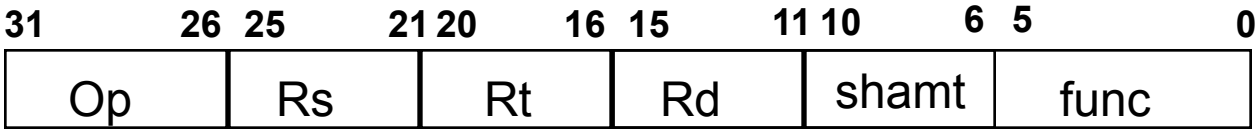
MIPS Integer Registers

- Registers
  - fast memory, part of the CPU.
  - smaller number
  - named in instruction
- 31 x 32-bit GPRs
  - R0 always holds 0
  - Often use symbols $0,$1,$2,… instead of r0,r1,r2…
- Other Registers
  - 32 x 32-bit FP regs (paired DP)
  - 32-bit HI, LO, PC

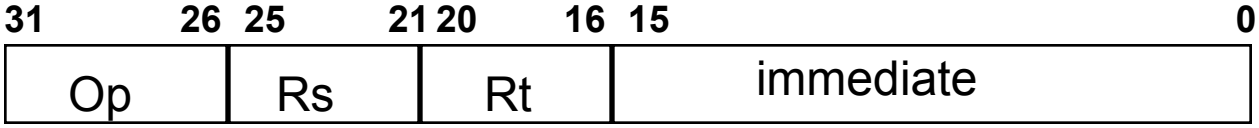


MIPS Instruction Formats

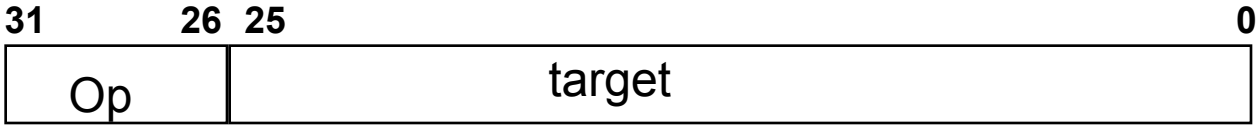
R-type: Register-Register



I-type: Register-Immediate



J-type: Jump / Call

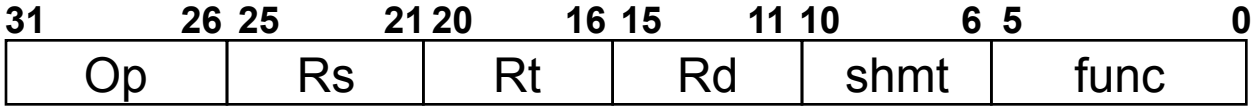


Terminology

Op = opcode

Rs, Rt, Rd = register specifier

R Type: <OP> rd, rs, rt



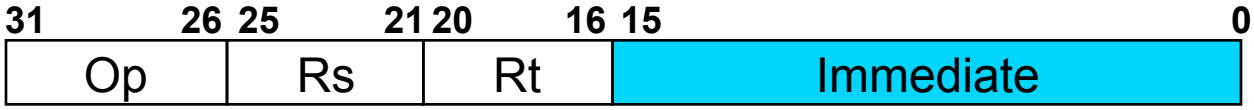
op	**a 6-bit operation code.**
rs	**a 5-bit source register.**
rt	**a 5-bit target (source) register.**
rd	**a 5-bit destination register.**
shmt	**a 5-bit shift amount.**
func	**a 6-bit function field.**

Operand Addressing: Register direct

Example: ADD $1, $2, $3 # $1 = $2 + $3

op	rs	rt	rd	shmt	func
000000	00010	00011	00001	00000	100000

I-Type <op> rt, rs, immediate



Immediate: 16 bit value

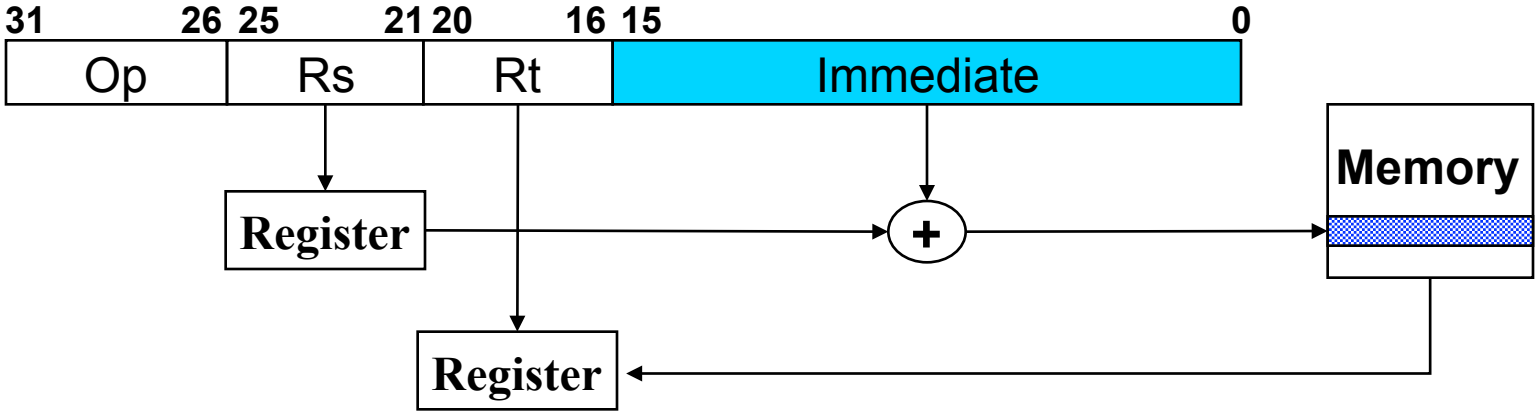
Operand Addressing: Register Direct and Immediate

Add Immediate Example

addi $1, $2, 100 # $1 = $2 + 100

op	rs	rt	immediate
001000	00010	00001	0000 0000 0110 0100

I-Type <op> rt, rs, immediate



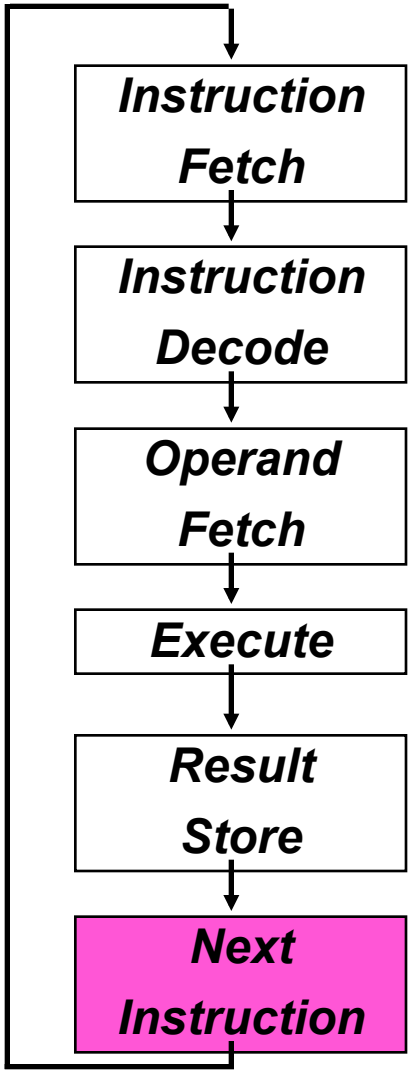
Memory Addressing: Base + Index

Load Word Example

lw $1, 100($2) # $1 = Mem[$2+100]

op	rs	rt	immediate
100011	00010	00001	0000 0000 0110 0100

Successor Instruction – Branches




```
main()
{
   int x,y,same;  // $0 == 0 always
   x = 43;                    // addi $1, $0, 43
   y = 2;                     // addi $2, $0, 2
   same = 0;                  // addi $3, $0, 0
   if (x == y)
         same = 1;            // execute only if x == y
                              // addi $3, $0, 1
}
```

The Program Counter (PC)

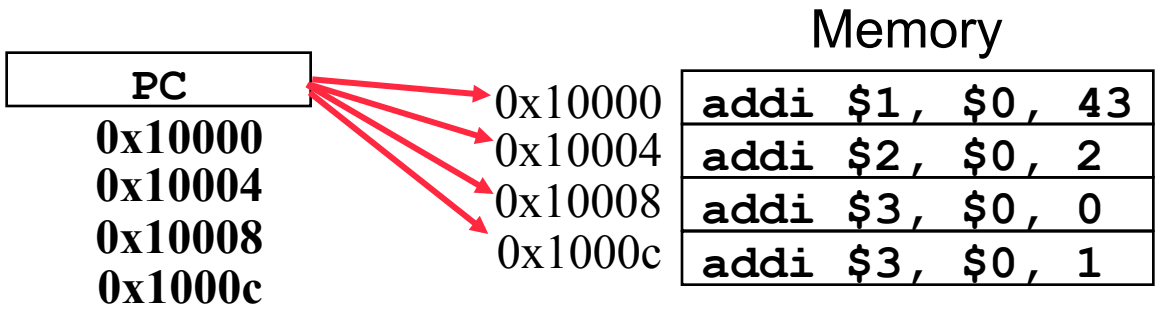
- Special register (PC) that points to instructions
- Contains memory address

- Instruction fetch is
  - inst = mem[pc]

- To fetch next sequential instruction PC = PC + ?
  - Size of instruction?

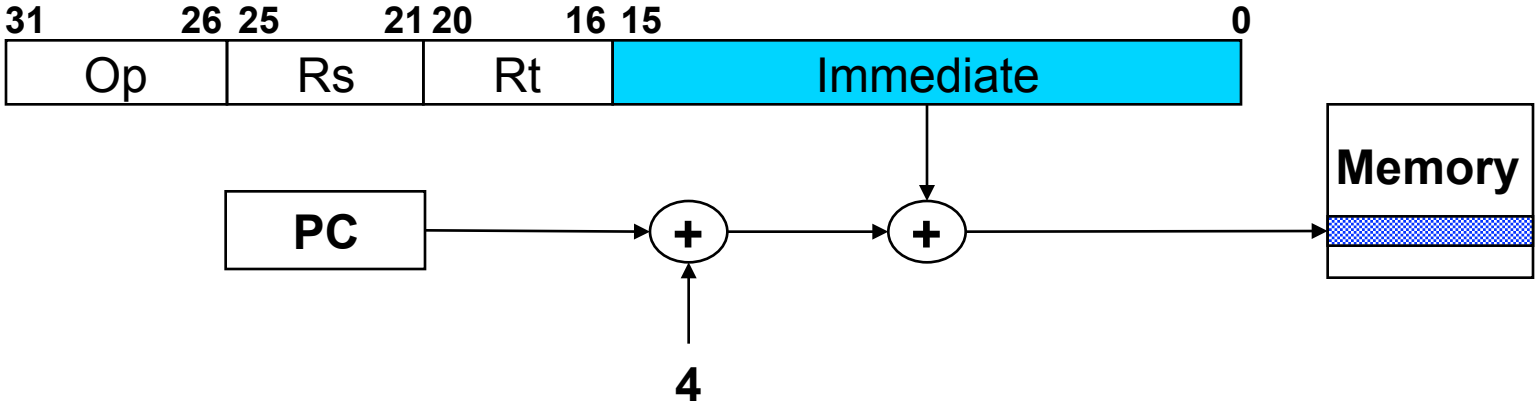
The Program Counter (PC)

```
x = 43;          // addi $1, $0, 43
y = 2;           // addi $2, $0, 2
same = 0;        // addi $3, $0, 0
if (x == y)
   same = 1;   // addi $3, $0, 1 execute if x == y
```



Suppose PC automatically increments to next instruction
Clearly, this is not correct
We cannot always execute both 0x10008 and 0x1000c

I-Type <op> rt, rs, immediate



PC relative addressing

Branch Not Equal

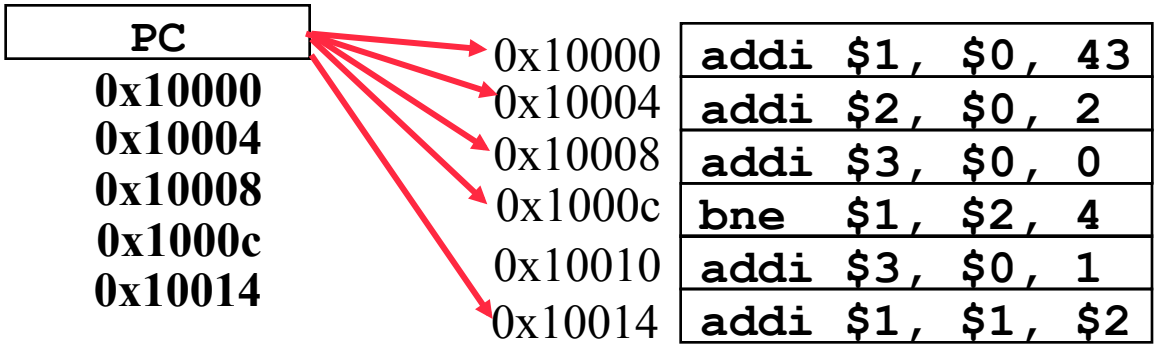
bne $1, $2, 100 # If ($1!= $2) goto [PC+4+100]

+4 because we increment sequentially by default

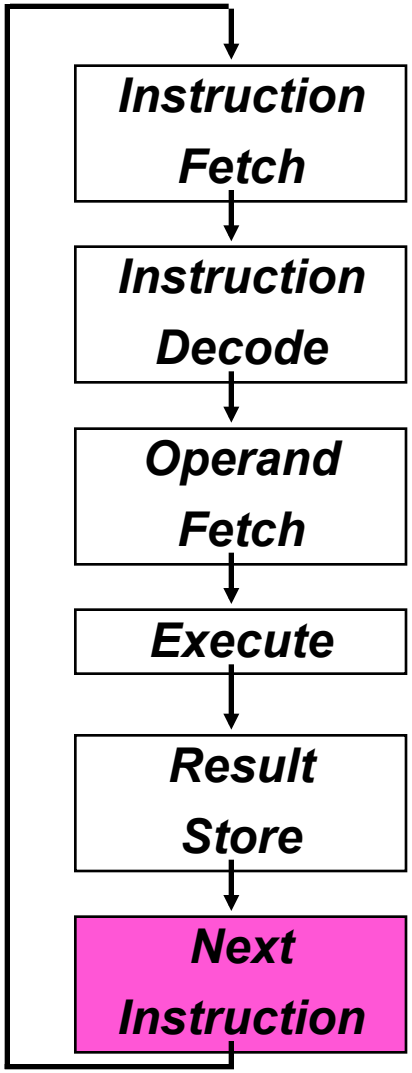
op	rs	rt	immediate
000101	00001	00010	0000 0000 0110 0100

The Program Counter (PC)

```
x = 43;          // addi $1, $0, 43
y = 2;           // addi $2, $0, 2
same = 0;        // addi $3, $0, 0
if (x == y)
   same = 1;     // addi $3, $0, $1 execute if x == y
x = x + y;       // addi $1, $1, $2
```



Successor Instruction – Function Calls




```
int equal(int a1, int a2) {
   int tsame;
   tsame = 0;
   if (a1 == a2)
        tsame = 1;    // only if a1 == a2
   return(tsame);
}
main()
{
   int x,y,same;          // r0 == 0 always
   x = 43;                // addi $1, $0, 43
   y = 2;                 // addi $2, $0, 2
   same = equal(x,y);     // need to call function
   // other computation
}
```

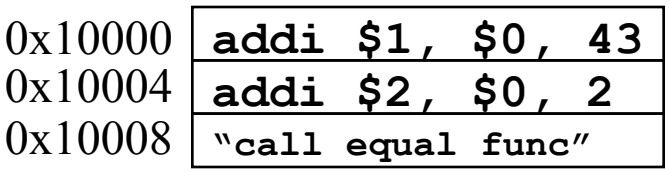
The Program Counter (PC)

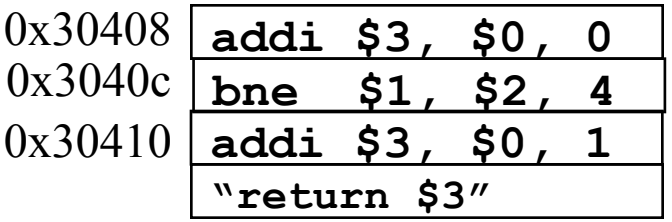
- Branches are limited to 16 bit immediate
  - 2^16 = 4K instructions

```
x = 43; // addi $1, $0, 43
y = 2;  // addi $2, $0, 2
same = equal(x,y);
```

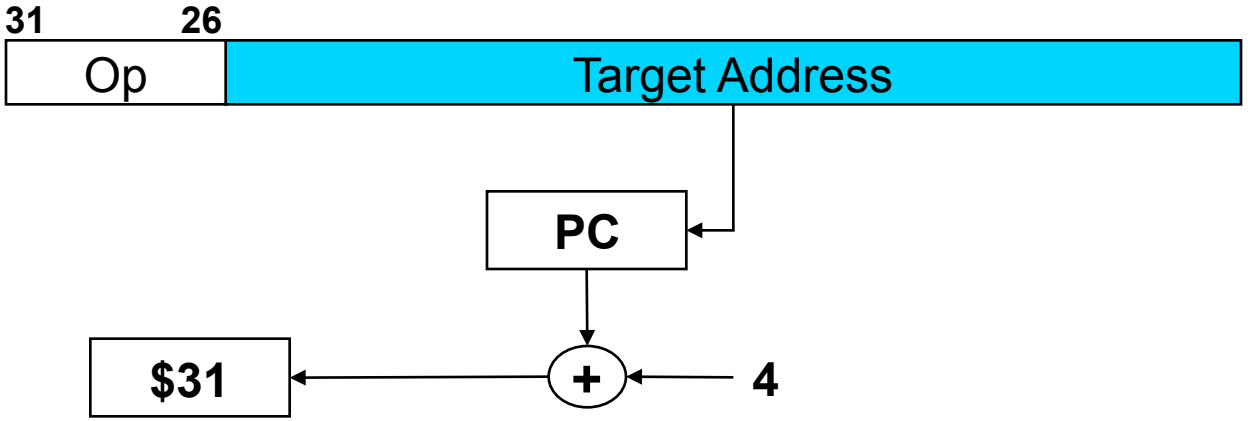
- Big programs?
  - Linked list 258 x86 instructions
  - With C library > 138,000 x86 instructions

- Support to two steps
  (1) Call function – JAL
  (2) Return – JR





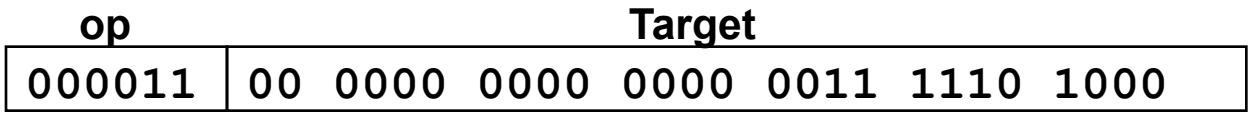
J-Type: <OP> target



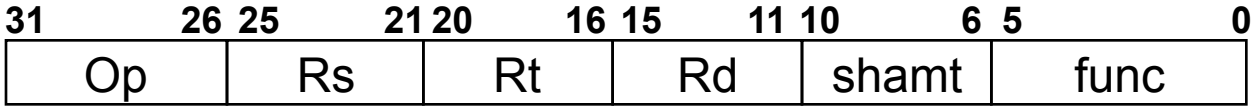
Jump and Link Example

JAL 1000 # PC<- 1000, $31<-PC+4

$31 set as return address

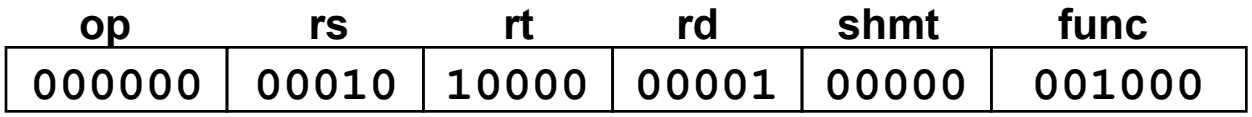


R Type: <OP> rd, rs, rt



Jump Register Example

jr $31 # PC <- $31



Instructions for Procedure Call and Return

```
int equal(int a1, int a2) {
   int tsame;
   tsame = 0;
   if (a1 == a2)
         tsame = 1;
         return(tsame);
}
main()
{
   int x,y,same;
   x = 43;
   y = 2;
   same = equal(x,y);
   // other computation
}
```

0x10000	addi $1, $0, 43
0x10004	addi $2, $0, 2
0x10008	jal 0x30408
0x1000c	??

0x30408	addi $3, $0, 0
0x3040c	bne $1, $2, 4
0x30410	addi $3, $0, 1
0x30414	jr $31

PC	$31
0x10000	??
0x10004	??
0x10008	??
0x30408	0x1000c
0x3040c	0x1000c
0x30410	0x1000c
0x30414	0x1000c
0x1000c	0x1000c

MIPS Arithmetic Instructions

Instruction	Example	Meaning	Comments
add	add $1,$2,$3	$1 = $2 + $3	3 operands
subtract	sub $1,$2,$3	$1 = $2 – $3	3 operands
add immediate	addi $1,$2,100	$1 = $2 + 100	+ constant
add unsigned	addu $1,$2,$3	$1 = $2 + $3	3 operands
subtract unsigned	subu $1,$2,$3	$1 = $2 – $3	3 operands
add imm. unsign.	addiu $1,$2,100	$1 = $2 + 100	+ constant
multiply	mult $2,$3	Hi, Lo = $2 x $3	64-bit signed product
multiply unsigned	multu $2,$3	Hi, Lo = $2 x $3	64-bit unsigned product
divide	div $2,$3	Lo = $2 ÷ $3, Hi = $2 mod $3	Lo = quotient, Hi = remainder
divide unsigned	divu $2,$3	Lo = $2 ÷ $3, Hi = $2 mod $3	Unsigned quotient Usigned remainder
Move from Hi	mfhi $1	$1 = Hi	Used to get copy of Hi
Move from Lo	mflo $1	$1 = Lo	Used to get copy of Lo

MIPS Logical Instructions

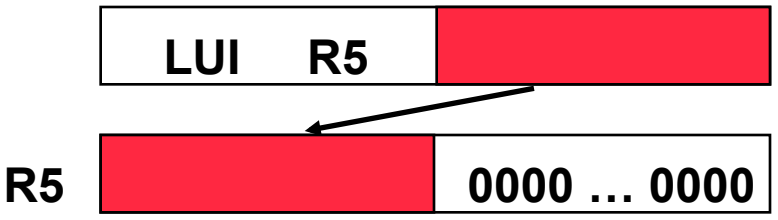
Instruction	Example	Meaning	Comment
and	and $1,$2,$3	$1 = $2 & $3	Bitwise AND
or	or $1,$2,$3	$1 = $2 | $3	Bitwise OR
xor	xor $1,$2,$3	$1 = $2 ⊕ $3	Bitwise XOR
nor	nor $1,$2,$3	$1 = ~($2 | $3)	Bitwise NOR
and immediate	andi $1,$2,10	$1 = $2 & 10	Bitwise AND reg, const
or immediate	ori $1,$2,10	$1 = $2 | 10	Bitwise OR reg, const
xor immediate	xori $1, $2,10	$1 = ~$2 &~10	Bitwise XOR reg, const
shift left logical	sll $1,$2,10	$1 = $2 << 10	Shift left by constant
shift right logical	srl $1,$2,10	$1 = $2 >> 10	Shift right by constant
shift right arithm.	sra $1,$2,10	$1 = $2 >> 10	Shift right (sign extend)
shift left logical	sllv $1,$2,$3	$1 = $2 << $3	Shift left by var
shift right logical	srlv $1,$2, $3	$1 = $2 >> $3	Shift right by var
shift right arithm.	srav $1,$2, $3	$1 = $2 >> $3	Shift right arith. by var

Note: MIPS Assembler has pseudo instructions available (e.g., NOT A == A XOR 1)

MIPS Data Transfer Instructions

Instruction	Comment
SW R3, 500(R4)	Store word
SH R3, 502(R2)	Store half
SB R2, 41(R3)	Store byte
LW R1, 30(R2)	Load word
LH R1, 40(R3)	Load halfword
LHU R1, 40(R3)	Load halfword unsigned
LB R1, 40(R3)	Load byte
LBU R1, 40(R3)	Load byte unsigned
LUI R1, 40	Load Upper Immediate (16 bits shifted left by 16)

Why do we need LUI?



MIPS Compare and Branch

Compare and Branch

beq	rs, rt, offset	if R[rs] == R[rt] then PC-relative branch
bne	rs, rt, offset	<>

Compare to Zero and Branch

blez	rs, offset	if R[rs] <= 0 then PC-relative branch
bgtz	rs, offset	>
bltz	rs, offset	<
bgez	rs, offset	>=
bltzal	rs, offset	if R[rs] < 0 then branch and link (into R 31)
bgeal	rs, offset	>=

- Remaining set of compare and branch take two instructions
- Almost all comparisons are against zero!

MIPS jump, branch, compare instructions

Instruction	Example	Meaning
branch on equal	beq $1,$2,100	if ($1 == $2) go to PC+4+100
	Equal test; PC relative branch	
branch on not eq.	bne $1,$2,100	if ($1!= $2) go to PC+4+100
	Not equal test; PC relative	
set on less than	slt $1,$2,$3	if ($2 < $3) $1=1; else $1=0
	Compare less than; 2's comp.	
set less than imm.	slti $1,$2,100	if ($2 < 100) $1=1; else $1=0
	Compare < constant; 2's comp.	
set less than uns.	sltu $1,$2,$3	if ($2 < $3) $1=1; else $1=0
	Compare less than; natural numbers	
set l. t. imm. uns.	sltiu $1,$2,100	if ($2 < 100) $1=1; else $1=0
	Compare < constant; natural numbers	
jump	j 10000	go to 10000
	Jump to target address	
jump register	jr $31	go to $31
	For switch, procedure return	
jump and link	jal 10000	$31 = PC + 4; go to 10000
	For procedure call	

Signed vs Unsigned Comparison

R1= 0…00 0000 0000 0000 0001
R2= 0…00 0000 0000 0000 0010
R3= 1…11 1111 1111 1111 1111

Execute these instructions:

```
slt  r4,r2,r1
slt  r5,r3,r1
sltu r6,r2,r1
sltu r7,r3,r1
```

What are values of registers r4 - r7? Why?

r4 = ; r5 = ; r6 = ; r7 = ;

Summary

- MIPS has 5 categories of instructions
  - Arithmetic, Logical, Data Transfer, Conditional Branch, Unconditional Jump
- 3 Instruction Formats

Next Time

- Assembly Programming

Reading

- An overview of SPIM and the MIPS32 instruction set @ http://spimsimulator.sourceforge.net/HP_AppA.pdf
- MIPS Quick Ref (PDF in docs section of web site)