ECE 550D
Fundamentals of Computer Systems and Engineering

Fall 2023

Instruction Set Architectures (ISAs) and MIPS

Xin Li & Dawei Liu
Duke Kunshan University

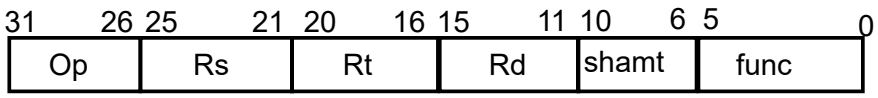
Slides are derived from work by
Andrew Hilton, Tyler Bletsch and Rabih Younes (Duke)

Last time…

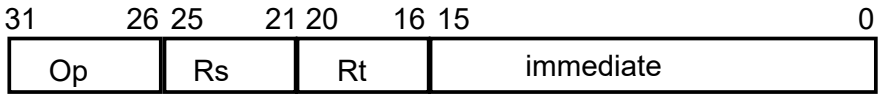
- Who can remind us what we did last time?
  - MIPS ISA and Assembly Programming
  - More on Assembly Programming today

MIPS Instruction Formats

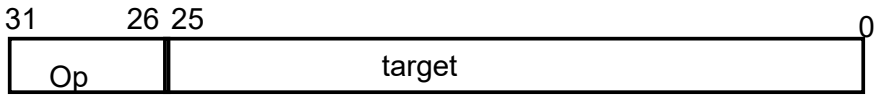
R-type: Register-Register



I-type: Register-Immediate



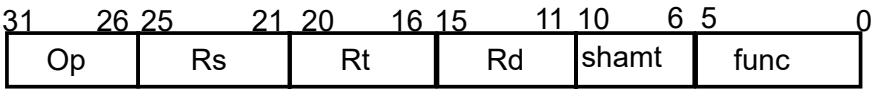
J-type: Jump / Call



Terminology
Op = opcode
Rs, Rt, Rd = register specifier

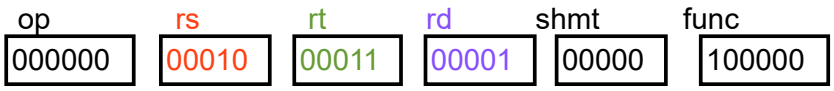
R Type: <OP> rd, rs, rt

R-type: Register-Register



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.

Example: add $1, $2, $3



Executing Add

Address	Instruction	Insn Val
1000	add $8, $4, $6	0086 4020
1004	add $5, $8, $7	0107 2820
1008	add $6, $6, $6	00C6 3020
100C	…	…
1010	…	…

Register	Value
$0	0000 0000
$1	1234 5678
$2	C001 D00D
$3	1BAD F00D
$4	9999 9999
$5	9999 999C
$6	0000 0002
$7	0000 0002
$8	9999 999A
$9	0000 0000
$10	0000 0000
…	…
PC	0000 100C

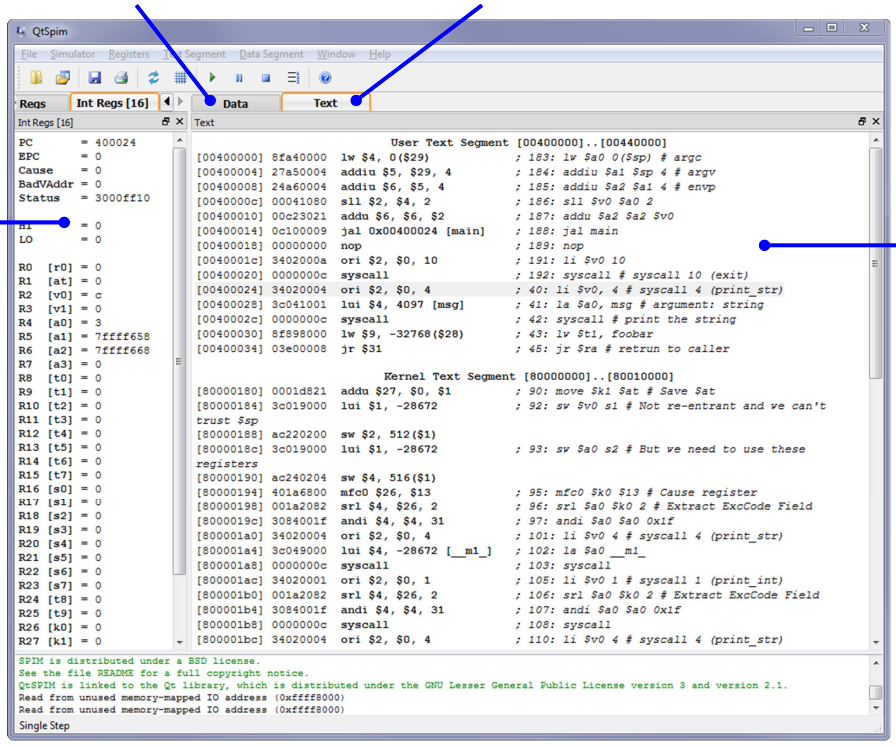
qtspim: Shows you similar state

Data memory is in this tab

Your program is in the "text" region of memory

Registers

Code



Other Similar Instructions

- `sub $rDest, $rSrc1, $rSrc2`
- `mul $rDest, $rSrc1, $rSrc2` (pseudo-insn)
- `div $rDest, $rSrc1, $rSrc2` (pseudo-insn)
- `and $rDest, $rSrc1, $rSrc2`
- `or  $rDest, $rSrc1, $rSrc2`
- `xor $rDest, $rSrc1, $rSrc2`
- …

- Don't need to remember every insn
- Will provide insn reference on tests

Pseudo Instructions

- Some "instructions" are pseudo-instructions
  - Actually assemble into 2 instructions:

- `mul $1, $2, $3` is really

```
mul $2, $3
mflo $1
```

- mul takes two srcs, writes special regs lo and hi
- mflo moves from lo into dst reg

What if I want to add a constant?

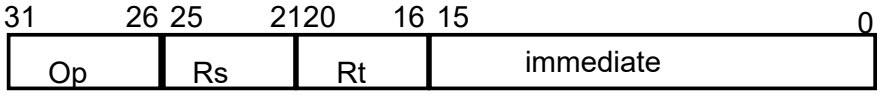
- Suppose I need to do
  - x = x + 1

  - Idea one: Put 1 in a register, use add
    - Problem: How to put 1 in a register?
  - Idea two: Have instruction that adds an **immediate**
    - Note: also solves problem in idea one

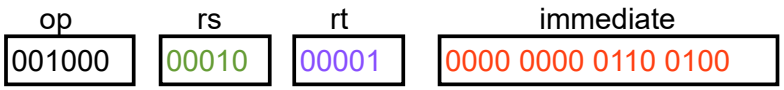
I-Type <op> rt, rs, immediate

- Immediate: 16 bit value

I-type: Register-Immediate



Add Immediate Example
addi $1, $2, 100



Using addi to put a const in register

- Can use addi to put a constant into a register:
  - x = 42;
  - Can be done with
  - addi $7, $0, 42
  - Because $0 is always 0.

- Common enough it has its own pseudo-insn:
  - `li $7, 42`
  - Stands for load immediate, works for 16-bit immediate

Many insns have Immediate Forms

- Add is not the only one with an immediate form
  - `andi`, `ori`, `xori`, `sll`, `srl`, `sra`, …

- No subi
  - Why not?

- No muli or divi
  - Though some ISAs have them

Assembly programming something "useful"

- Consider the following C fragment:

```
int tempF = 87;
int a = tempF - 32;
a = a * 5;
int tempC = a / 9;
```

```
li $3, 87

addi $4, $3, -32

li $6, 5
mul $4, $4, $6

li $6, 9
div $5, $4, $6
```

- Lets write assembly for it
  - First, need registers for our variables:
    - `tempF = $3`
    - `a = $4`
    - `tempC = $5`

Accessing Memory

- MIPS is a "load-store" ISA
  - Who can remind us what that means?
  - Only load and store insns access memory
  - (and that is all those isns do)
- Contrast to x86, which allows
  - add reg = (memory location) + reg
- Or even
  - add (memory location) = (memory location) + reg

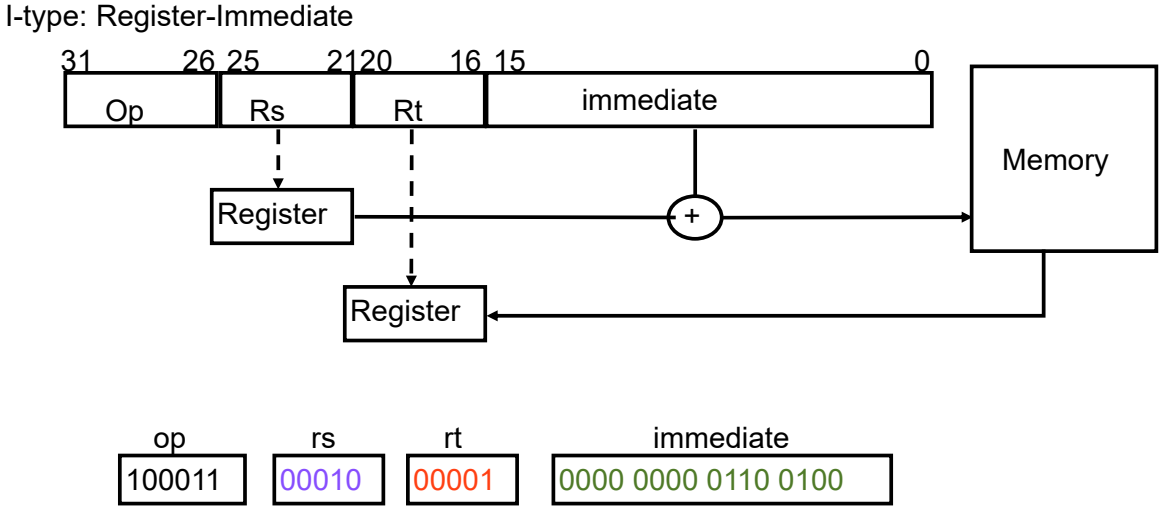
Back to the Simple, Running Example

- assume $6=1004=address of variable x in C code example
- and recall that 1008=address of variable y in C code example

```
loop:  lw $1, Memory[1004] → lw $1, 0($6) # put val of x in $1
       lw $2, Memory[1008] → lw $2, 4($6) # put val of y in $2
       add $3, $1, $2
       add $4, $4, $3
       j loop
```

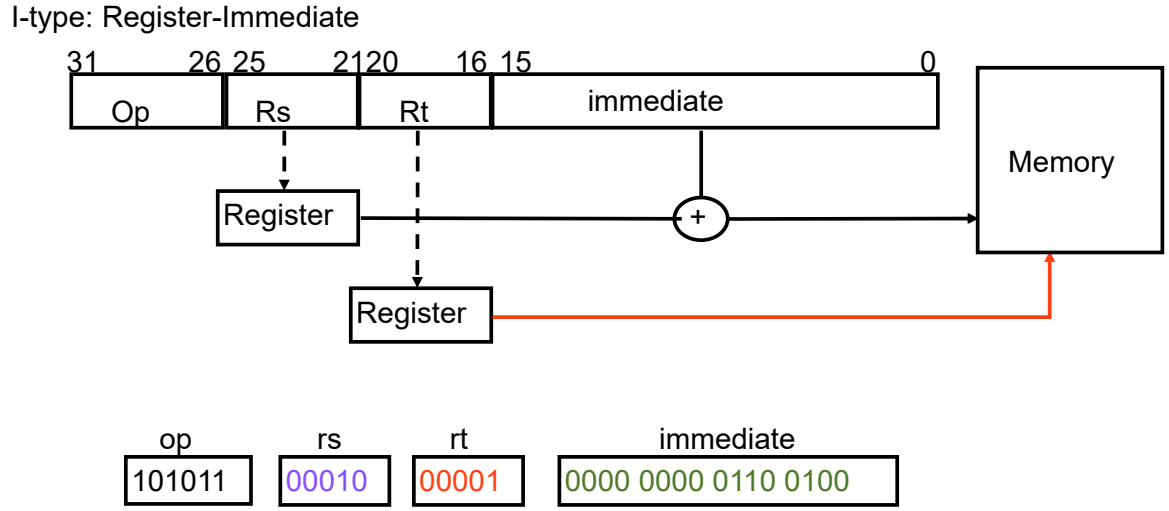

I-Type <op> rt, rs, immediate

- Load Word Example
- `lw $1, 100($2)` # $1 = Mem[$2+100]



I-Type <op> rt, rs, immediate

- Store Word Example
- `sw $1, 100($2)` # Mem[$2+100] = $1



Data sizes / types

- Loads and Stores come in multiple sizes
  - Reflect different data types
- The w in lw/sw stands for "word" (= 32 bits)
  - Can also load bytes (8 bits), half words (16 bits)
  - Smaller sizes have signed/unsigned forms
  - We will distribute a list of all instructions for your reference

C and assembly: loads/stores

- int x
- int * p
- int ** q
- ...
- x = *p;
- **q = x;
- p = *q;
- p[4] = x;

```
# in $1
# in $2
# in $3
...
lw $1, 0($2)
lw $4, 0($3)
sw $1, 0($4)
lw $2, 0($3)
sw $1, 16($2)
```


Executing Memory Ops

Address	Value
1000	lw $1, 0($2)
1004	lw $4, 4($3)
1008	sw $1, 8($4)
100C	lw $2, 0($3)
1010	...
...	...
8000	F00D F00D
8004	C001 D00D
8008	1234 4321
8010	4242 4242
8014	0000 8000

Register	Value
$0	0000 0000
$1	1234 5678
$2	0000 8004
$3	0000 8010
$4	9999 9999
$5	9999 999C
$6	0000 0002
$7	0000 0002
$8	9999 999A
$9	0000 0000
$10	0000 0000
$11	0000 0000
$12	0000 0000
...	...
PC	0000 1000

Executing Memory Ops

Address	Value
1000	lw $1, 0($2)
1004	lw $4, 4($3)
1008	sw $1, 8($4)
100C	lw $2, 0($3)
1010	…
…	…
8000	F00D F00D
8004	C001 D00D
8008	1234 4321
8010	4242 4242
8014	0000 8000

Register	Value
$0	0000 0000
$1	C001 D00D
$2	0000 8004
$3	0000 8010
$4	9999 9999
$5	9999 999C
$6	0000 0002
$7	0000 0002
$8	9999 999A
$9	0000 0000
$10	0000 0000
$11	0000 0000
$12	0000 0000
…	…
PC	0000 1004

Executing Memory Ops

Address	Value
1000	lw $1, 0($2)
1004	lw $4, 4($3)
1008	sw $1, 8($4)
100C	lw $2, 0($3)
1010	…
…	…
8000	F00D F00D
8004	C001 D00D
8008	1234 4321
8010	4242 4242
8014	0000 8000

Register	Value
$0	0000 0000
$1	C001 D00D
$2	0000 8004
$3	0000 8010
$4	0000 8000
$5	9999 999C
$6	0000 0002
$7	0000 0002
$8	9999 999A
$9	0000 0000
$10	0000 0000
$11	0000 0000
$12	0000 0000
…	…
PC	0000 1008

Executing Memory Ops

Address	Value
1000	lw $1, 0($2)
1004	lw $4, 4($3)
1008	sw $1, 8($4)
100C	lw $2, 0($3)
1010	…
…	…
8000	F00D F00D
8004	C001 D00D
8008	C001 D00D
8010	4242 4242
8014	0000 8000

Register	Value
$0	0000 0000
$1	C001 D00D
$2	0000 8004
$3	0000 8010
$4	0000 8000
$5	9999 999C
$6	0000 0002
$7	0000 0002
$8	9999 999A
$9	0000 0000
$10	0000 0000
$11	0000 0000
$12	0000 0000
…	…
PC	0000 100C

Executing Memory Ops

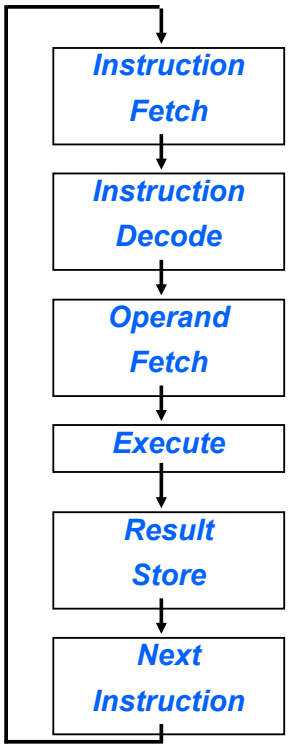
Address	Value
1000	lw $1, 0($2)
1004	lw $4, 4($3)
1008	sw $1, 8($4)
100C	lw $2, 0($3)
1010	…
…	…
8000	F00D F00D
8004	C001 D00D
8008	C001 D00D
8010	4242 4242
8014	0000 8000

Register	Value
$0	0000 0000
$1	C001 D00D
$2	4242 4242
$3	0000 8010
$4	0000 8000
$5	9999 999C
$6	0000 0002
$7	0000 0002
$8	9999 999A
$9	0000 0000
$10	0000 0000
$11	0000 0000
$12	0000 0000
…	…
PC	0000 1010

Making control decision

- Control constructs—decide what to do next:

```
if (x == y) {
…
}
else {
…
}
…
while (z < q) {
…
}
```

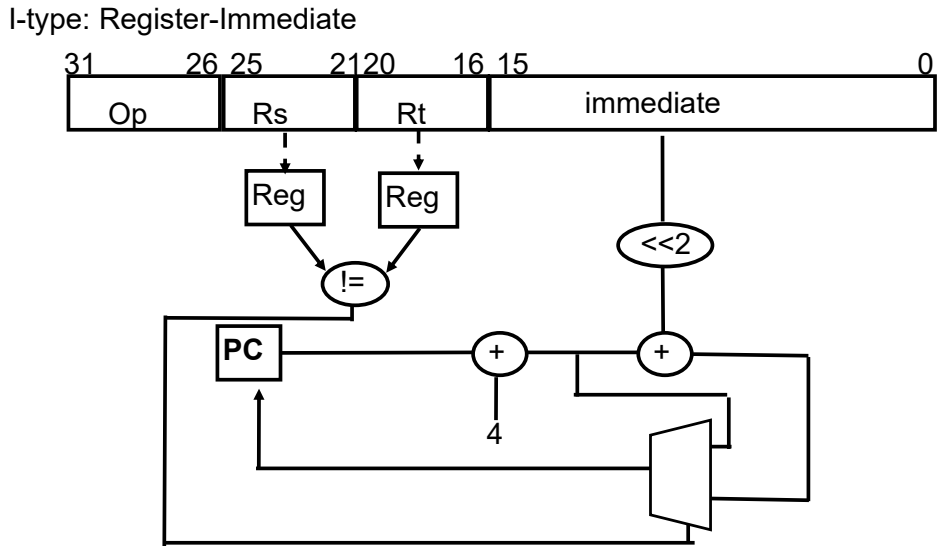


The Program Counter (PC)

- Special register (PC) that points to instructions
- Contains memory address (like a pointer)
- Instruction fetch is
  - inst = mem[pc]
- So far, have fetched sequentially: PC= PC + 4
  - Assumes 4 byte insns
  - True for MIPS
  - X86: variable size (nasty)
- May want to specify non-sequential fetch
  - Change PC in other ways

I-Type <op> rt, rs, immediate

- PC relative addressing
- Branch Not Equal Example
- bne $1, $2, 100 # If ($1!= $2) goto [PC+4+400]



MIPS Compare and Branch

- Compare and Branch
  - `beq  rs, rt, offset`
  - `bne  rs, rt, offset`
- Compare to zero and Branch
  - `blez  rs, offset`
  - `bgtz  rs, offset`
  - `bltz  rs, offset`
  - `bgez  rs, offset`
- Also pseudo-insns for unconditional branch (`b`)

MIPS jump, branch, compare instructions

- Inequality to something other than 0: require 2 insns
  - Conditionally set reg, branch if not zero or if zero

- `slt $1,$2,$3` $1 = ($2 < $3) ? 1 : 0
  - Compare less than; signed 2's comp.

- `slti $1,$2,100` $1 = ($2 < 100) ? 1 : 0
  - Compare < constant; signed 2's comp.

- `sltu $1,$2,$3` $1 = ($2 < $3) ? 1 : 0
  - Compare less than; unsigned

- `sltiu $1,$2,100` $1 = ($2 < 100) ? 1 : 0
  - Compare < constant; unsigned

- `beqz $1,100` if ($1 == 0) go to PC+4+400
  - Branch if equal to 0

- `bnez $1,100` if ($1 != 0) go to PC+4+400
  - Branch if not equal to 0

Signed v.s. Unsigned Comparison

- $1= 0…00 0000 0000 0000 0001
- $2= 0…00 0000 0000 0000 0010
- $3= 1…11 1111 1111 1111 1111
- After executing these instructions:

```
slt  $4,$2,$1
slt  $5,$3,$1
sltu $6,$2,$1
sltu $7,$3,$1
```

- What are values of registers $4 - $7? Why?

  $4 = 0 ; $5 = 1 ; $6 = 0 ; $7 = 0 ;

C and Assembly with branches

```
int x;                          //assume x in $1
int y;                          //assume y in $2
int z;                          //assume z in $3
…                               …
if (x != y) {                   beq $1,$2, 2
  z = z + 2;                    addi $3, $3, 2
}                               b 1
else {
  z = z -4;                     addi $3, $3, -4
}
```


Labels

- Counting insns?
  - Error prone
  - Tricky: pseudo-insns
  - Un-maintainable
- Better: let assembler count
  - Use a label
  - Symbolic name for target
  - Assembler computes offset

```
//assume x in $1
//assume y in $2
//assume z in $3
…
   beq $1, $2, L_else
   addi $3, $3, 2
   b L_end
L_else:
   addi $3, $3, -4
L_end:
```

Summary

- MIPS ISA and Assembly Programming
- Continue on next lecture…