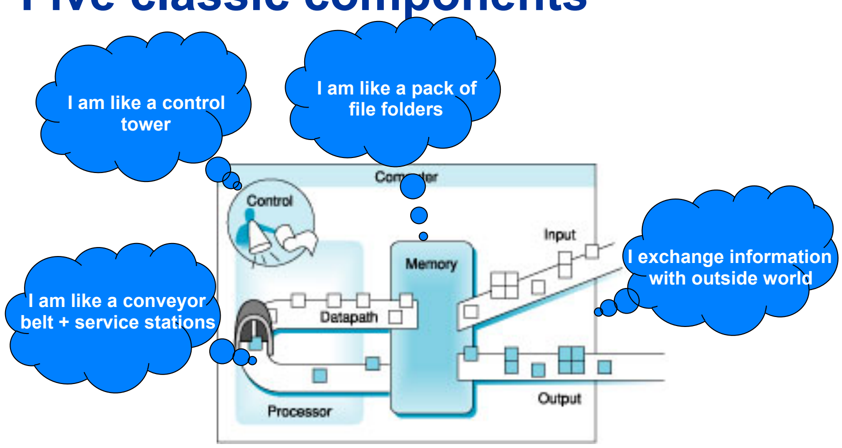
CS/COE0447: Computer Organization and Assembly Language

Chapter 2

modified by Bruce Childers
original slides by Sangyeun Cho

Dept. of Computer Science
University of Pittsburgh

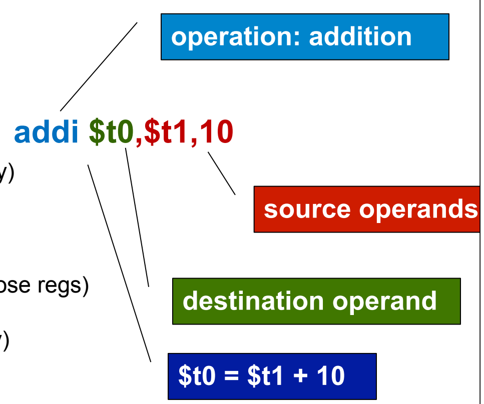
Five classic components



MIPS operations and operands

- Operation specifies **what function** to perform by the instruction
- Operand specifies **what quantity** to use with the instruction

- MIPS operations
  - Arithmetic (integer/floating-point)
  - Logical (AND, OR, etc)
  - Shift (moves bits around)
  - Compare (equality test)
  - Load/store (get/put stuff in memory)
  - Branch/jump (make decisions)
  - System control and coprocessor
- MIPS operands
  - Registers (one of 32 general-purpose regs)
  - Fixed registers (e.g., HI/LO)
  - Memory location (place in memory)
  - Immediate value (constant)



MIPS arithmetic

- **<op> <r_{target}>, <$r_{source1}$>, <$r_{source2}$>**

- All arithmetic instructions have 3 operands
  - Operand order in notation is fixed; target (destination) first
  - Two source registers and one target (destination) register
  - Operands are either *2 registers* or *1 register + 1 immediate* (constant)
  - Destination is *always* a register
- Examples
  - add $s1, $s2, $s3 # $s1 ⇐ $s2 + $s3
  - sub $s4, $s5, $s6 # $s4 ⇐ $s5 – $s6

MIPS registers

Name	32 bits	32 bits	Name	32 bits
$zero	r0	r16	$s0	HI
$at	r1	r17	$s1	LO
$v0	r2	r18	$s2	
$v1	r3	r19	$s3	
$a0	r4	r20	$s4	
$a1	r5	r21	$s5	
$a2	r6	r22	$s6	
$a3	r7	r23	$s7	
$t0	r8	r24	$t8	
$t1	r9	r25	$t9	
$t2	r10	r26	$k0	
$t3	r11	r27	$k1	
$t4	r12	r28	$gp	
$t5	r13	r29	$sp	
$t6	r14	r30	$fp	
$t7	r15	r31	$ra	PC

General-Purpose Registers
Just a table, or array, of temporary locations referred to by a number (register number)

Special-Purpose Registers

General-purpose registers (GPRs)

- The name GPR implies that **all these registers can be used as operands in instructions**
- Still, **conventions and limitations** exist to keep GPRs from being used arbitrarily (from the PRM)
  - **$0, termed $zero, always has a value of "0"**
  - **$31, termed $ra (return address), is reserved for storing the return address for subroutine call/return**
  - Register usage and related software conventions are typically summarized in "application binary interface" (ABI) – important when writing system software such as an assembler or a compiler
- 32 GPRs in MIPS
  - Are they sufficient?

Special-purpose registers

- **HI/LO registers** used to store result from multiplication
- **PC register** (program counter)
  - Always keeps the pointer to the current program execution point; instruction fetching occurs at the address in PC
  - Not directly visible and manipulated by programmers in MIPS
- Other instruction set architectures
  - May not have HI/LO; use GPRs to store the result of multiplication
  - May allow storing to PC to make a jump

Instruction encoding

- Instructions are ***encoded*** in ***binary numbers***
  - Assembler translates assembly programs into binary numbers
  - Machine (processor) decodes binary numbers to figure out what the original instruction is
  - MIPS has a fixed, 32-bit instruction encoding
- Encoding should be done in a way that decoding is easy
- MIPS instruction formats
  - **R-format**: arithmetic instructions
  - **I-format**: data transfer/arithmetic/branch instructions
  - **J-format**: jump instruction format (changes program counter)
  - (**FI-/FR-format**: floating-point instruction format)

MIPS instruction formats

Name	Fields *bit 31*					*bit 0*	Comments
Field Size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits
R-format	op (opcode)	rs	rt	rd	shamt	funct	Arithmetic/logic instruction format
I-format	op (opcode)	rs	rt	Immediate/address			Data transfer, branch, imm. format
J-format	op (opcode)	target address					Jump instruction format

Instruction encoding example

add $t0,$t1,$t8

$t8 is register 24
rt = 11000 (5 bits)

$t1 is register 9
rs = 01001 (5 bits)

$t0 is register 8
rd = 01000 (5 bits)

operation is "addition"
opcode = 000000 (6 bits)
funct = 100000 (6 bits)

shamt is unused
shamt = 00000 (5 bits)

Resulting encoded instruction:

Op	Rs	Rt	Rd	Shamt	Funct

00000001001110000100000000100000

Dealing with immediate

Name	Fields				Comments
I-format	op	rs	rt	16-bit immediate	Transfer, branch, immediate format

- Many operations involve small "immediate" value
  - a = a + 1
  - b = b – 4
  - c = d & 0xff
- Example instructions
  - addi $s3, $s2, 1 # $s3 ⇐ $s2 + 1
  - addi $s4, $s1, -4 # $s4 ⇐ $s1 + (-4)
  - andi $s5, $s0, 0xff # $s5 ⇐ $s0 & 0x000000ff
- Immediate is pos/neg up to 15 bits (15 bit value with 1 bit "sign")
- li $reg,immediate # $s3 ⇐ 0xFDECBA98 (up to 32 bits)

Interacting with the OS

- We need the OS's help!!!
  - How to print a number? (output)
  - How to read a number? (input)
  - How to terminate (halt) a program?
  - How to open, close, read, write a file?
  - These are operating system "services"
- Special instruction: **syscall**
  - A "software interrupt" to invoke OS for an action (to do a *service*)
  - Need to **indicate the service to perform** (e.g., print vs. terminate)
  - May also need to **pass an argument value** (e.g., number to print)

A few useful syscalls

- **syscall** takes a service ID (number) sent to OS in $v0
  - *<load arguments>*
  - *<set service id in $v0>*
  - syscall
- Print integer
  - $v0=1, $a0=integer to print
- Read integer
  - $v0=5, after syscall, $v0 holds the integer read from keyboard
- Print string
  - $v0=4, $a0=memory address of string to print (null terminated)
- Exit (halt)
  - $v0=10, no argument
- See MARS docs for more!!! Also, attend recitation.

Example: Print 100d

```
li  $a0,100    # value to print
li  $v0,10     # print int service
syscall        # call OS
```


Example: First Asm. Program!

Program should do the following:

1. Ask user for a number, X
2. Add 100 to X
3. Print the result (X+100)
4. Exit

What do we need?

syscall to input, output number, exit program

add instruction for X + 100

load immediate

Example: First Asm. Program!

```
li      $v0,5              # read integer, X
syscall                    # returns X in $v0
addi  $a0,$v0,100          # $a0 = $v0 + 100
li      $v0,1              # print integer in $a0
syscall                    # invoke OS
li      $v0,10             # exit program
syscall
```


Example: Second Asm. Program!

- Let's clean this up a bit.
  - We should prompt the user to ask for a number.
  - We should print a prompt with the output.
- We need to use ***strings*** in the assembly program.
  - The strings are data!
  - Specify string name, string type, and string value
- Data is specified in special part of program: "data section"
- Data has general format:

  name*:** ***.type ***data-values***

  allowed *types* are: asciiz, word, byte, etc.

```
        .data                                              DATA
msg1:   .asciiz               "Enter a value:\n"
msg2:   .asciiz               "Sum of value and 100:\n"
        .text
        li      $v0,4         # prompt user (print string)
        la      $a0,msg1      # indicate the message
        syscall
        li      $v0,5         # read integer, X             CODE
        syscall
        addi    $s0,$v0,100   # $s0 = X + 100
        li      $v0,4         # output message
        la      $a0,msg2      # indicate the message
        syscall
        li      $v0,1         # print integer
        move    $a0,$s0       # value to print
        syscall
        li      $v0,10        # exit program
        syscall
```


Logic instructions

Name	Fields						Comments
R-format	op	rs	rt	rd	shamt	funct	Logic instruction format

- Bit-wise logic operations
- <op> <r_{target}>, <$r_{source1}$>, <$r_{source2}$>
- Examples
  - and $s3, $s2, $s1 # $s3 ⇐ $s2 & $s1
  - or $t3, $t2, $t1 # $t3 ⇐ $t2 | $t1
  - nor $s3, $t2, $s1 # $s3 ⇐ ~($t2 | $s1)

    note: nor $s3,$t2,$0 is $s3 ⇐ !($t2) (not of $t2)
  - xor $s3, $s2, $s1 # $s3 ⇐ $s2 ^ $s1

    note: xor produces 1 *iff* one of the operands is 1

Logic Instructions with Immediates

- Logic instructions have I-format (small immediate) versions
  - andi $s0,$s1,0xff00
  - ori $s0,$s1,0x0ff0
  - xori $s0,$s1,0xf00f
  - nori $s0,$s1,0xffff
- Upper bits (bits 31..16) are set to 0s by instruction
  - E.g., 0xff00 is really 0x0000ff00
  - This operation is known as "zero extension"

Handling long immediate number

- li allows loading large immediates (> 16 bits)
  - ***Pseudo-operation***: Assembler "converts" to *actual* machine instructions
- Consider: li $s3,0xAA55CC33
- Converted to two instructions:
  - lui $s3, 1010 1010 0101 0101b

$s3	1010101001010101	0000000000000000

- Then we fill the low-order 16 bits
  - ori $s3, $s3, 1100 1100 0011 0011b

$s3	1010101001010101	1100110000110011

Shift instructions

Name	Fields						Comments
R-format	op	NOT USED	rt	rd	shamt	funct	shamt is "shift amount"

- Bits change their positions inside a word
- <op> <r_{target}> <r_{source}> <shift_amount>
- Examples
  - sll $s3, $s4, 4 # $s3 ⇐ $s4 << 4
  - srl $s6, $s5, 6 # $s6 ⇐ $s5 >> 6
- Shift amount can be in a register ("shamt" is not used)
- Shirt right arithmetic (sra) keeps the sign of a number
  - sra $s7, $s5, 4

Let's try it in MARS!!!! (mips6.asm)