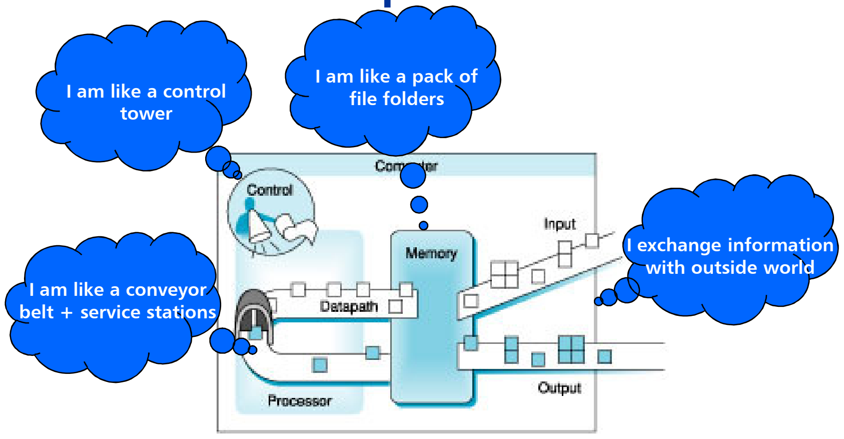
CS/COE0447: Computer Organization and Assembly Language

Chapter 2

modified by Bruce Childers
original slides by Sangyeun Cho

Dept. of Computer Science
University of Pittsburgh

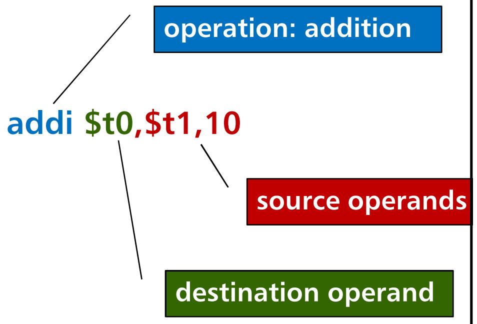
Five classic components


I am like a control tower
I am like a pack of file folders
I exchange information with outside world
I am like a conveyor belt + service stations
Computer
Control
Memory
Input
Datapath
Processor
Output

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
  - Shift
  - Compare
  - Load/store
  - Branch/jump
  - System control and coprocessor
- MIPS operands
  - Registers
  - Fixed registers (e.g., HI/LO)
  - Memory location
  - Immediate value



MIPS arithmetic

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

- All arithmetic instructions have 3 operands
  - Operand order in notation is fixed; target first
  - Two source registers and one target or destination register
  - Operands are 2 registers or 1 register + 1 immediate
  - Destination is a register

- Examples
  - add $s1, $s2, $s3 # $s1 ⇐ $s2 + $s3
  - sub $s4, $s5, $s6 # $s4 ⇐ $s5 – $s6

MIPS registers

	32 bits	32 bits		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   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 are used for storing result from multiplication operations
- PC (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 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/jump instructions
  - J-format: jump instruction format (changes program counter)
  - (FI-/FR-format: floating-point instruction format)

MIPS instruction formats

Name	*bit 31* Fields					*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	rs	rt	rd	shamt	funct	Arithmetic/logic instruction format
I-format	op	rs	rt	address/immediate			Data transfer, branch, immediate format
J-format	op	target address					Jump instruction format

CS/CoE0447: Computer Organization and Assembly Language

University of Pittsburgh

Instruction encoding example

add $t0,$t1,$t8

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

$t1 is register r9
rs = 01001 (5 bits)

$t0 is register r8
rd = 01000 (5 bits)

op 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

CS/CoE0447: Computer Organization and Assembly Language

University of Pittsburgh

Instruction encoding example

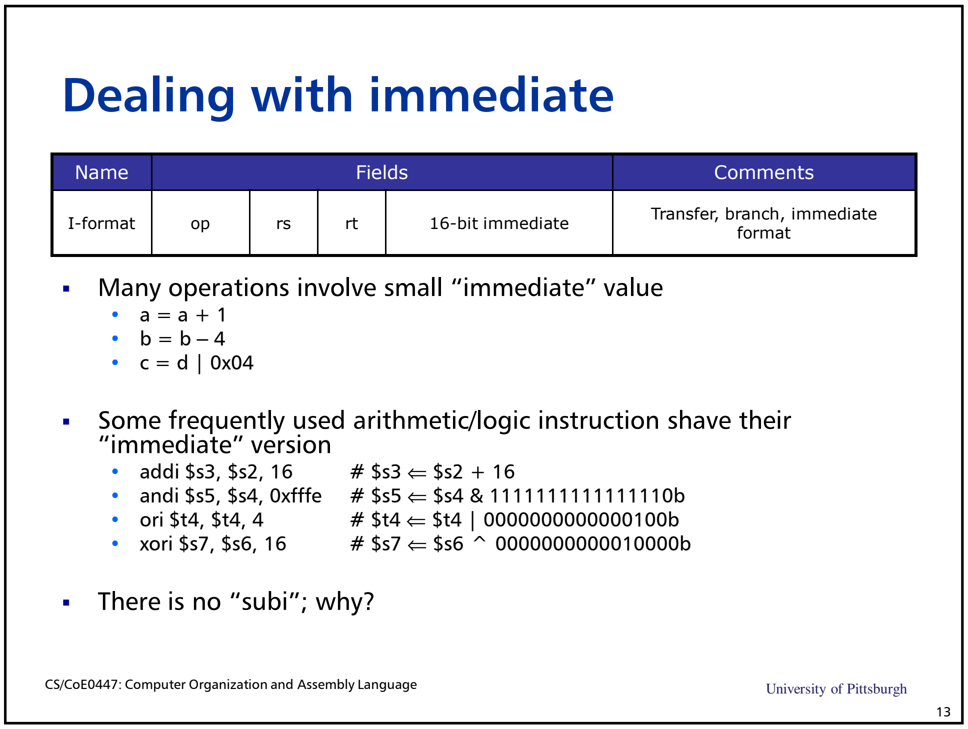
Instruction	Format	op	rs	rt	rd	shamt	funct	immediate
add	R	000000	reg	reg	reg	00000	100000	NA
sub	R	000000	reg	reg	reg	00000	100010	NA
addi (add immediate)	I	001000	reg	reg	NA	NA	NA	constant
lw (load word)	I	100011	reg	reg	NA	NA	NA	address offset
sw (store word)	I	101011	reg	reg	NA	NA	NA	address offset

Logic instructions

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

- Bit-wise logic operations
- <op> $<r_{target}>$ $<r_{source1}>$ $<r_{source2}>$

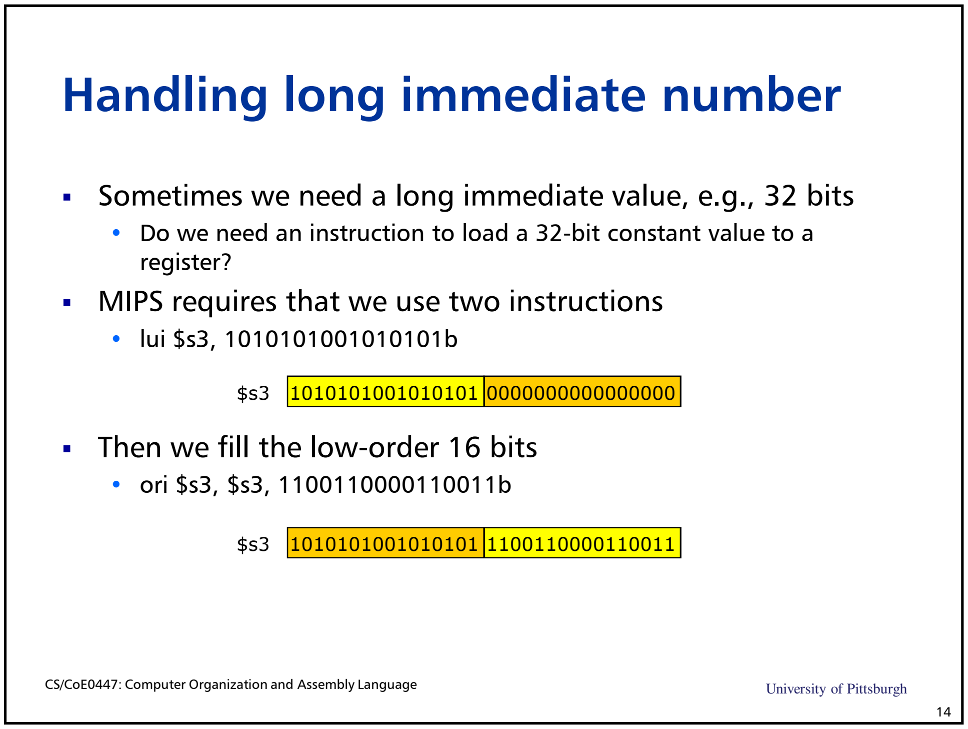
- Examples
  - and $s3, $s2, $s1 # $s3 ⇐ $s2 & $s1
  - or $t3, $t2, $t1 # $t3 ⇐ $t2 | $t1
  - nor $s3, $t2, $s1 # $s3 ⇐ ~($t2 | $s1)
  - xor $s3, $s2, $s1 # $s3 ⇐ $s2 ^ $s1



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 | 0x04
- Some frequently used arithmetic/logic instruction shave their "immediate" version
  - addi \$s3, \$s2, 16 # \$s3 ⇐ \$s2 + 16
  - andi \$s5, \$s4, 0xfffe # \$s5 ⇐ \$s4 & 1111111111111110b
  - ori \$t4, \$t4, 4 # \$t4 ⇐ \$t4 | 0000000000000100b
  - xori \$s7, \$s6, 16 # \$s7 ⇐ \$s6 ^ 0000000000010000b
- There is no "subi"; why?



Handling long immediate number

- Sometimes we need a long immediate value, e.g., 32 bits
  - Do we need an instruction to load a 32-bit constant value to a register?
- MIPS requires that we use two instructions
  - lui \$s3, 1010101001010101b

\$s3	1010101001010101	0000000000000000

- Then we fill the low-order 16 bits
  - ori \$s3, \$s3, 1100110000110011b

\$s3	1010101001010101	1100110000110011

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) in $v0
- 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 an integer

```
# example as C program code
#    int a;
#    a = 10 + 8;
#    print(“%d”, a);
# code below carries out the above
li          $t0,10        # $t0 is a, $t0=10
addi        $t0,$t0,8     # $t0=10+8
li          $v0,1         # print service
add         $a0,$t0,$0    # pass value in $t0 to service
syscall                   # invoke OS to do service
li          $v0,10        # terminate program service
syscall                   # invoke OS to do service
```

Let’s try it in MARS!!!!

Example: More useful output

```
  li          $t0,10               # put 10 into a
  addi        $t0,$t0,8            # do the add with 8
  li          $v0,4                # print string service
  la          $a0,msg              # load string
  syscall
  li          $v0,1                # print integer ervice
  add         $a0,$t0,$0
  li          $v0,10               # terminate program
  syscall
  # message to print before the number
.data
  msg:        .asciiz      “Sum of 10 + 8 is\n”
```

Let’s try it in MARS!!!!

Example: Add 10 + x?

```
   li         $v0,4               # print prompt
   la         $a0,x_msg
   syscall
   li         $v0,5               # read integer service
   syscall
   move       $t0,$v0             # number read in $v0
   addi       $t0,$t0,10          # add number with 10
   li         $v0,4               # print result prompt
   la         $a0,msg
   syscall
   li         $v0,1               # print the sum
   move       $a0,$t0
   li         $v0,10              # terminate program
   syscall
.data
   x_msg:     .asciiz      "Number x to add?\n"
   msg:       .asciiz      "Sum of 10 + x is\n"
```

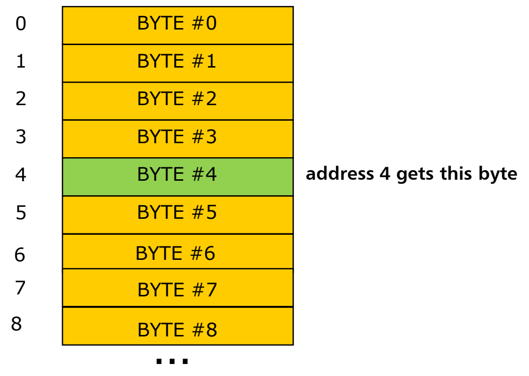
Let's try it in MARS!!!!

Memory transfer instructions

- Also called *memory access* instructions
- Only two types of instructions
  - Load: move data from memory to register
    - e.g., lw $s5, 4($t6)   # $s5 ⇐ memory[$t6 + 4]
  - Store: move data from register to memory
    - e.g., sw $s7, 16($t3)   # memory[$t3+16] ⇐ $s7
- In MIPS (32-bit architecture) there are memory transfer instructions for
  - 32-bit word: "int" type in C
  - 16-bit half-word: "short" type in C
  - 8-bit byte: "char" type in C

Memory view

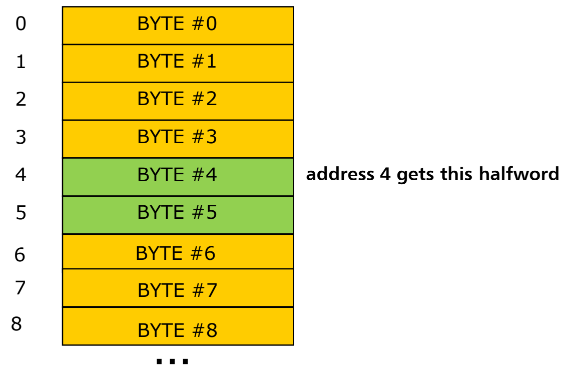
- Memory is a large, single-dimension 8-bit (byte) array with an address to each 8-bit item (“byte address”)
- A ***memory address is just an index into the array***



- loads and stores give the index (address) to access

Memory view

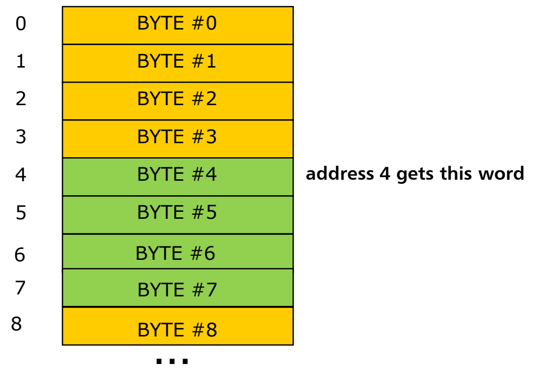
- Memory is a large, single-dimension 8-bit (byte) array with an address to each 8-bit item (“byte address”)
- A ***memory address is just an index into the array***



- loads and stores give the index (address) to access

Memory view

- Memory is a large, single-dimension 8-bit (byte) array with an address to each 8-bit item (“byte address”)
- A ***memory address is just an index into the array***



- loads and stores give the index (address) to access

Address calculation

- Memory address is specified with (**register, constant**) pair
  - Register to keep the ***base address***
  - Constant field to keep the ***offset*** from the base address
  - Address is, then (contents of register + offset)
  - The offset can be positive or negative
- Suppose base register $t0=64, then:

```
lw     $t0, 12($t1)         address = 64 + 12 = 76
lw     $t0, -12($t1)        address = 64 - 12 = 52
```

- MIPS uses this simple address calculation method; other architectures such as PowerPC and x86 support different methods

Machine code example

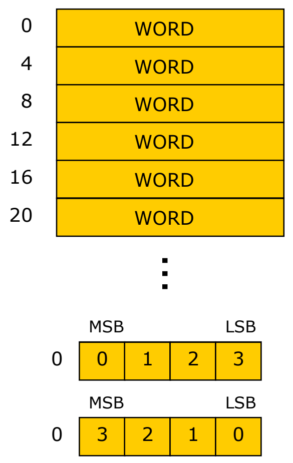
```
void swap(int v[], int k)
{
            int temp;
            temp = v[k];
            v[k] = v[k+1];
            v[k+1] = temp;
}
```

```
swap:
            sll         $t0, $a1, 2
            add         $t1, $a0, $t0
            lw          $t3, 0($t1)
            lw          $t4, 4($t1)
            sw          $t4, 0($t1)
            sw          $t3, 4($t1)
            jr          $ra
```

Let’s try it in MARS!!!!

Memory organization

- 32-bit byte address
  - 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
  - 2 least significant bits (LSBs) of an address are 0s
- Half words are aligned
  - LSB of an address is 0
- Addressing within a word
  - Which byte appears first and which byte the last?
  - Big-endian vs. little-endian



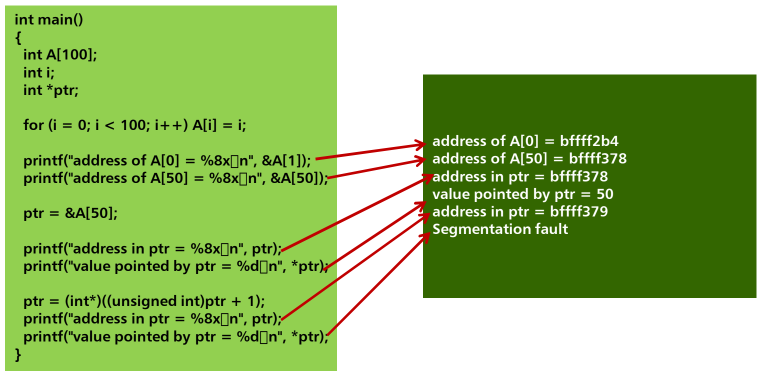
More on alignment

- A misaligned access
  - lw $s4, 3($t0)
- How do we define a word at address?
  - Data in byte 0, 1, 2, 3
    - If you meant this, use the address 0, not 3
  - Data in byte 3, 4, 5, 6
    - If you meant this, it is indeed misaligned!
    - Certain hardware implementation may support this; usually not
    - If you still want to obtain a word starting from the address 3 – get a byte from address 3, a word from address 4 and manipulate the two data to get what you want
- Alignment issue does not exist for byte access

0	0	1	2	3
4	4	5	6	7
8	8	9	10	11

⋮

More on alignment



```
int main()
{
  int A[100];
  int i;
  int *ptr;

  for (i = 0; i < 100; i++) A[i] = i;

  printf("address of A[0] = %8x\n", &A[1]);
  printf("address of A[50] = %8x\n", &A[50]);

  ptr = &A[50];

  printf("address in ptr = %8x\n", ptr);
  printf("value pointed by ptr = %d\n", *ptr);

  ptr = (int*)((unsigned int)ptr + 1);
  printf("address in ptr = %8x\n", ptr);
  printf("value pointed by ptr = %d\n", *ptr);
}
```

```
address of A[0] = bffff2b4
address of A[50] = bffff378
address in ptr = bffff378
value pointed by ptr = 50
address in ptr = bffff379
Segmentation fault
```