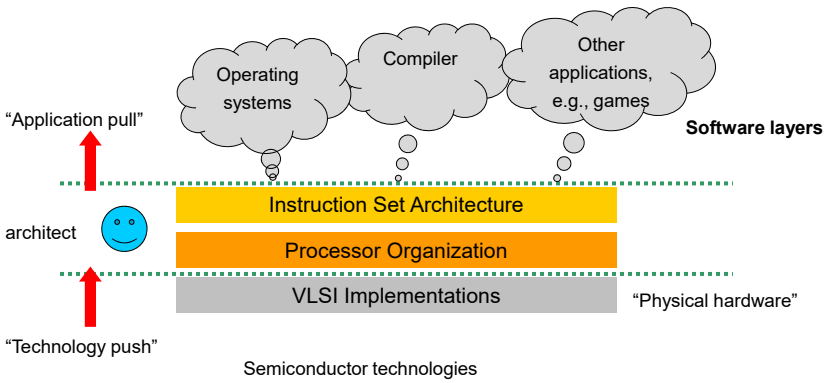
CS/COE1541: Introduction to Computer Architecture

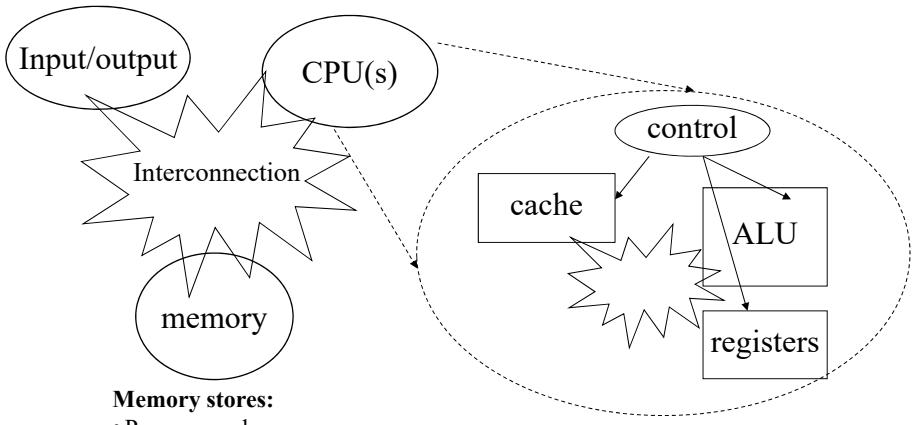
Dept. of Computer Science
University of Pittsburgh

http://www.cs.pitt.edu/~melhem/courses/1541p/index.html

Computer Architecture?



High level system architecture



Memory stores:
- Program code
- Data (static or dynamic)
- Execution stack

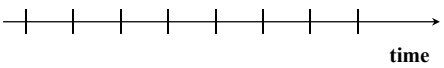
Important registers:
- Program counter (PC)
- Instruction register (IR)
- General purpose registers

Chapter 1 Review: Computer Performance

- **Response Time (latency)**
  - How long does it take for "a task" to be done?
- **Throughput**
  - Rate of completing the "tasks"
- **CPU execution time (our focus)**
  - doesn't count I/O or time spent running other programs
- **Performance = 1 / Execution time**
- **Important architecture evaluation metrics**
  - Traditional metrics: TIME and COST
  - More recent metrics: POWER and TEMPERATURE
  - In this course, we will mainly talk about time

Clock Cycles (from Section 1.6)

- Instead of using execution time in seconds, we can use number of cycles
  - Clock "ticks" separate different activities:



- cycle time = time between ticks = seconds per cycle
- clock rate (frequency) = cycles per second (1 Hz. = 1 cycle/sec)

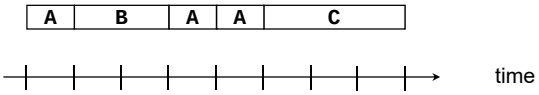
EX: A 2 GHz. clock has a cycle time of $\frac{1}{2 \times 10^9} = 0.5 \times 10^{-9}$ seconds.

- Time per program = cycles per program x time per cycle

$$\frac{\text{seconds}}{\text{program}} = \frac{\text{cycles}}{\text{program}} \times \frac{\text{seconds}}{\text{cycle}}$$

$$= \frac{\text{cycles}}{\text{Instruction}} \times \frac{\text{Instructions}}{\text{program}} \times \frac{\text{seconds}}{\text{cycle}}$$

Non-pipelined CPUs



- **Different instructions may take different number of cycles to execute (Why?)**

- **Cycles per instruction (CPI):**

Average number of cycles for executing an instruction (depends instruction mix)

Example: Assume that in a program:

20% of the instructions take one cycle (per instruction) to execute,
30% of the instructions take two cycles (per instruction) to execute and
50% of the instructions take three cycles (per instruction) to execute.

The CPI for the program = 0.2 * 1 + 0.3 * 2 + 0.5 * 3 = 2.3 cycles/instruction

Machine performance

$$\text{Execution time} = \text{CPI} \times \#\text{ of instructions} \times \text{Cycle time}$$

- Suppose we have two implementations of the same instruction set architecture (ISA).

  For some program,

  Machine A has a clock cycle time of 10 ns. and a CPI of 2.0
  Machine B has a clock cycle time of 20 ns. and a CPI of 1.2

  What machine is faster for this program (note that number of instructions in the program is the same)? `Answer: Machine A is faster`

Definitions:

# of instructions per cycle (IPC) = inverse of the CPI

MIPS ISA (from chapter 2)

MIPS assumes 32 CPU registers ($0, … , $31)

- All arithmetic instructions have 3 operands
- Operand order is fixed (destination first in assembly instruction)
- All operands of arithmetic operations are in registers
- Load and store instructions with simple memory addressing mechanism

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

MIPS code:
```
add $t0, $s1, $s2
add $s0, $t0, $s3
sub $s4, $s5, $s0
```

The compiler

t0, s1, s2, … are symbolic names of registers (translated to the corresponding numbers by the assembler).

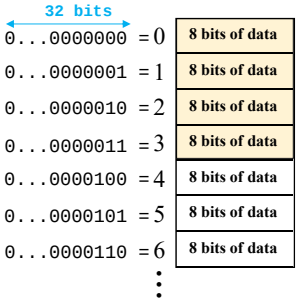
Register usage Conventions

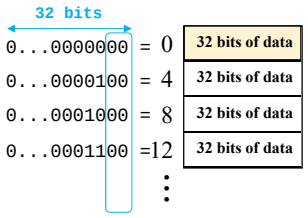
Name	Register number	Usage
$zero	0	the constant value 0
$v0-$v1	2-3	values for results and expression evaluation
$a0-$a3	4-7	arguments
$t0-$t7	8-15	temporaries
$s0-$s7	16-23	saved (correspond to program variables)
$t8-$t9	24-25	more temporaries
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address

Note: registers 26 and 27 are reserved for kernel use.

Memory Organization

- Viewed as a large, single-dimension array of bytes.
- A memory address is an index into the array
- "Byte addressing" means that the index points to a byte of memory.
- A word in MIPS is 32 bits long or 4 bytes (will not consider 64-bit versions)





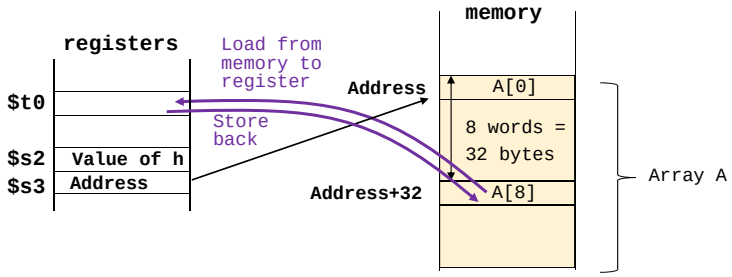
- Address space is 2^{32} bytes with byte addresses from 0 to 2^{32}-1
- 2^{30} words – the address of a word is the address of its first byte
  - the least 2 significant bits of a word address are always 00
  - Sometimes only 30 bits are used as a word address (drop the 00)

Load and Store Instructions

- **A load/store memory address = content of a register + a constant**

C code: `A[8] = h + A[8];`

MIPS code:
```
lw   $t0, 32($s3)    // load word
add  $t0, $s2, $t0
sw   $t0, 32($s3)    // store word
```



- Can you modify the above example to add h to all the elements A[0], … , A[8]?
- Need additional types of instructions?

Control instructions

- **Decision making instructions**
  - alter the control flow,
  - i.e., change the "next" instruction to be executed

- **MIPS conditional branch instructions:**

```
beq   $t0, $t1, label     // branch if $t0 and $t1 contain identical data
bne   $t0, $t1, label     // branch if $t0 and $t1 contain different data
```

  - one may assign a label to any instruction in assembly language

- **MIPS unconditional branch instruction:**

```
j   label
```

```
L1: beq $4,$5,L2
    sub ...
    ...
    ...
    j L3
L2: add ...
    ...
L3: ...
```

To summarize (short version of Figure 2.1):

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
	add	add $s1, $s2, $s3	$s1 = $s2 + $s3	Three operands; data in registers
Arithmetic	subtract	sub $s1, $s2, $s3	$s1 = $s2 - $s3	Three operands; data in registers
	add immediate	addi $s1, $s2, 100	$s1 = $s2 + 100	Used to add constants
	load word	lw $s1, 100($s2)	$s1 = Memory[$s2 + 100]	Word from memory to register
	store word	sw $s1, 100($s2)	Memory[$s2 + 100] = $s1	Word from register to memory
Data transfer	load byte	lb $s1, 100($s2)	$s1 = Memory[$s2 + 100]	Byte from memory to register
	store byte	sb $s1, 100($s2)	Memory[$s2 + 100] = $s1	Byte from register to memory
	load upper immediate	lui $s1, 100	$s1 = 100 * 2^{16}	Loads constant in upper 16 bits
	branch on equal	beq $s1, $s2, 25	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
Conditional	branch on not equal	bne $s1, $s2, 25	if ($s1 != $s2) go to PC + 4 + 100	Not equal test; PC-relative
branch	set on less than	slt $s1, $s2, $s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for beq, bne
	set less than immediate	slti $s1, $s2, 100	if ($s2 < 100) $s1 = 1; else $s1 = 0	Compare less than constant
	jump	j 2500	go to 10000 in current segment	Jump to target address
Uncondi-	jump register	jr $ra	go to $ra	For switch, procedure return
tional jump	jump and link	jal 2500	$ra = PC + 4; go to 10000	For procedure call

- Instruction operands:
  - Some instructions use 3 registers
  - Some instructions use 2 registers and a constant
  - Some instructions use one register
  - Some instructions use a constant

"C" program down to numbers

```
swap:
        muli    $2, $5, 4
        add     $2, $4, $2
        lw      $15, 0($2)
        lw      $16, 4($2)
        sw      $16, 0($2)
        sw      $15, 4($2)
        jr      $31
```

compiler

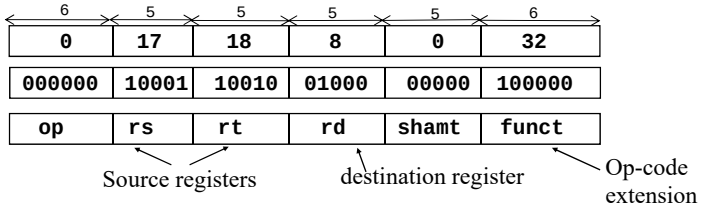
```
void swap(int v[], int k)
{
        int temp;
        temp = v[k];
        v[k] = v[k+1];
        v[k+1] = temp;
}
```

assembler

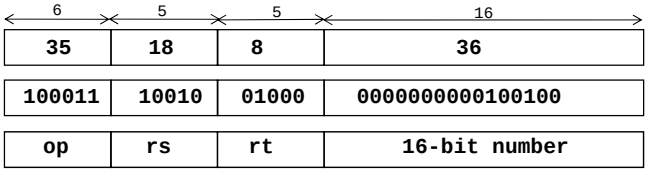
```
00000000101000010…
00000000000110000…
10001100011000100…
10001100111100100…
10101100111100100…
10101100011000100…
00000011111000000…
```

Machine Language instructions

- Instructions, like registers and data words, are also 32 bits long
- **R-type instruction format:** opcode rd, rs, rt
  - **Example**: add $t0, $s1, $s2 → add $8, $17, $18



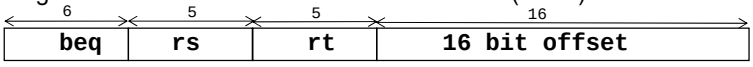
- **I-type instruction format:** opcode rt, I(rs)
  - **Examples**: `lw $t0, 36($s2)` // sw has a similar format

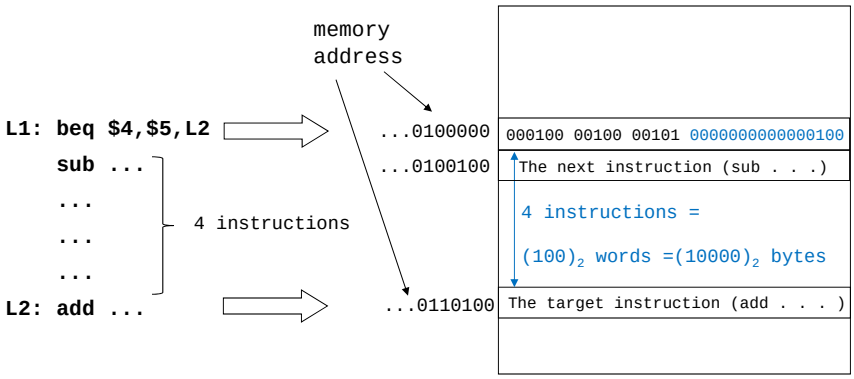


Branch instructions (I-type)

beq $4, $5, Label ===> put target address in PC if $4 == $5

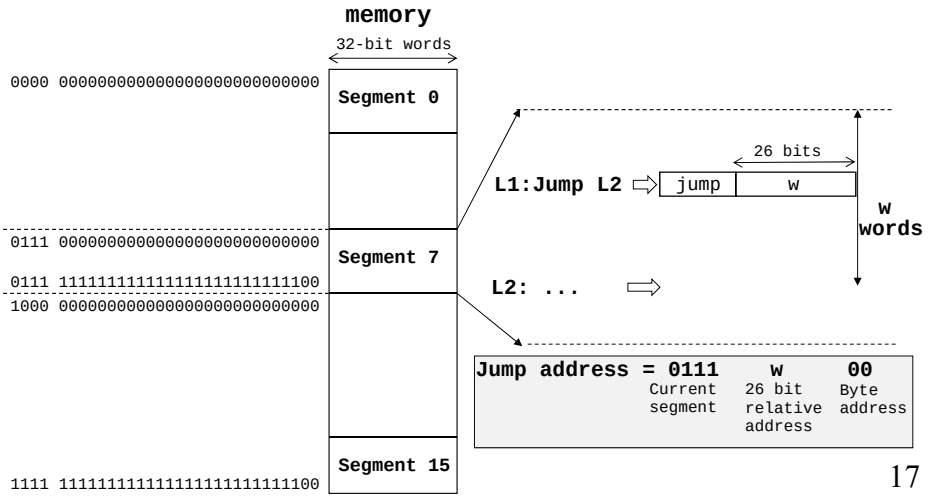
The address stored in the 16-bit address of a branch instruction is the address of the target instruction relative to the next instruction (offset).z





Jump instructions (J-type)

- In Jump instructions, address is relative to the 4 high order bits of PC+4
  - Memory is logically divided into 16 segments of 256 MB each.
  - address in Jump instruction = offset from the start of the current segment



Overview of MIPS

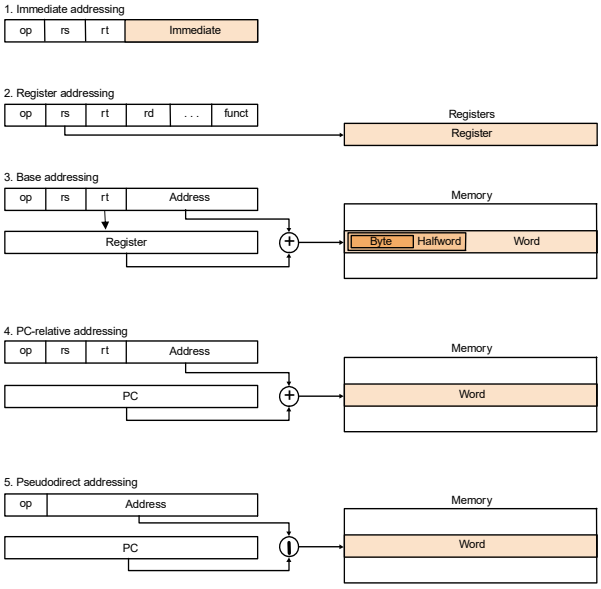
- only three instruction formats
-

R	op	rs	rt	rd	shamt	funct
I	op	rs	rt	16 bit constant/offset		
J	op	26 bit relative address				

  - Arithmetic and logic instructions use the **R-type** format
  - Memory and branch instructions *(lw, sw, beq, bne)* use the **I-type** format
  - Jump instructions use the **J-type** format

- Refer to section A.10 for complete specifications of MIPS instructions

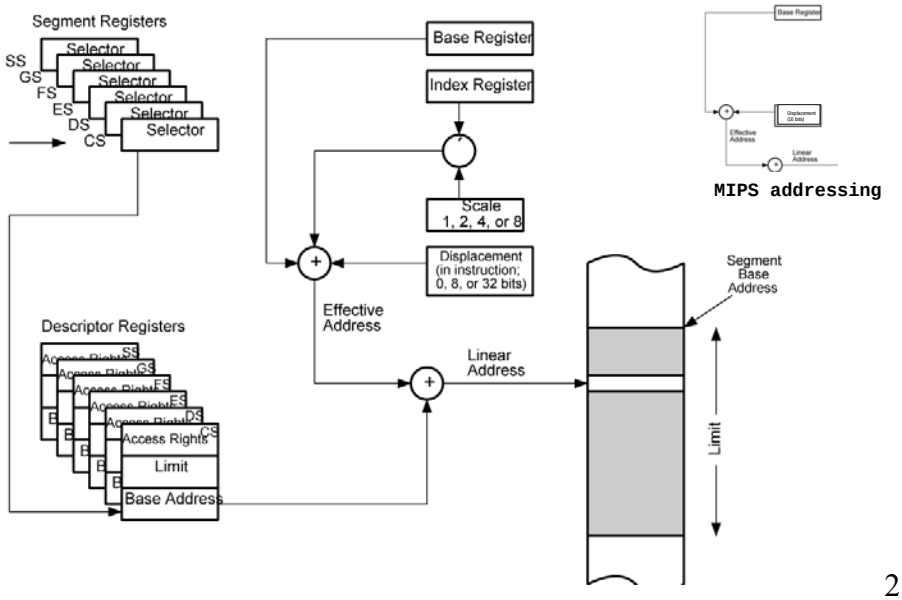
MIPS Addressing modes (operands' specification)



x86 Addressing Modes (A contrast)

- 12 addressing modes available to compute the effective address
  - Immediate (operand is in instruction – as in MIPS)
  - Register operand (operand in register – as in MIPS)
  - Displacement (the address of the operand is in the instruction
    - similar to the jump instruction in MIPS (without the 4 PC bits)
  - Relative (same as PC-relative in MIPS)
  - Base + displacement (address = content of a base register + displacement)
    - similar to base addressing in MIPS
  - Base (same as above with displacement = 0)
  - Scaled index with displacement (use an index rather than a base register)
  - Base with index and displacement (use two registers; base and index)
  - Base scaled index with displacement (multiply the index by a scale)
- Actual address (linear address) = effective address + base address (the beginning of the current segment).
- Instructions other than load/store can access memory

x86 Address Calculation



x86 Instruction Format (variable length instructions)