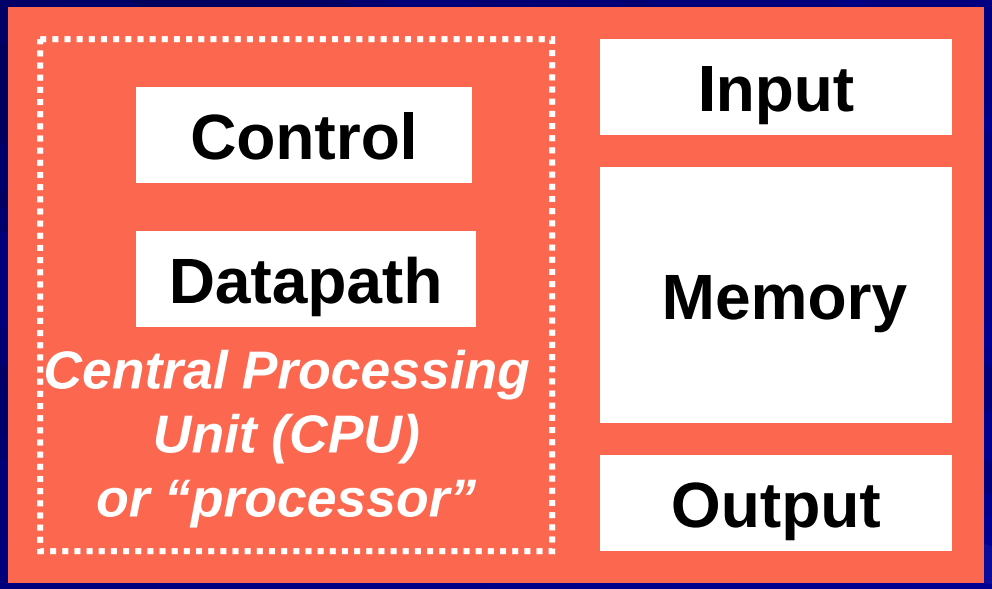
ELEC 5200-001/6200-001
Computer Architecture and Design
Spring 2019
Instruction Set Architecture (Chapter 2)

Christopher B. Harris
Assistant Professor
Department of Electrical and Computer Engineering
Auburn University, Auburn, AL 36849

(Adapted from slides by Vishwani D. Agrawal)

Designing a Computer



FIVE PIECES OF HARDWARE

Types of ISA

- Complex instruction set computer (CISC)
  - Many instructions (several hundreds)
  - An instruction takes many cycles to execute
  - Example: Intel x86, Pentium, Core I5
- Reduced instruction set computer (RISC)
  - Small set of instructions (typically 32)
  - Simple instructions, each executes in one clock cycle
    - ***REALLY? Well, almost.***
  - Effective use of pipelining
  - Example: ARM

On Two Types of ISA

- Brad Smith, “ARM and Intel Battle over the Mobile Chip’s Future,” Computer, vol. 41, no. 5, pp. 15-18, May 2008.

- Compare 3Ps:
  - Performance
  - Power consumption
  - Price

Simple Pipelining

Operations are broken up into stages and completed in assembly line fashion.

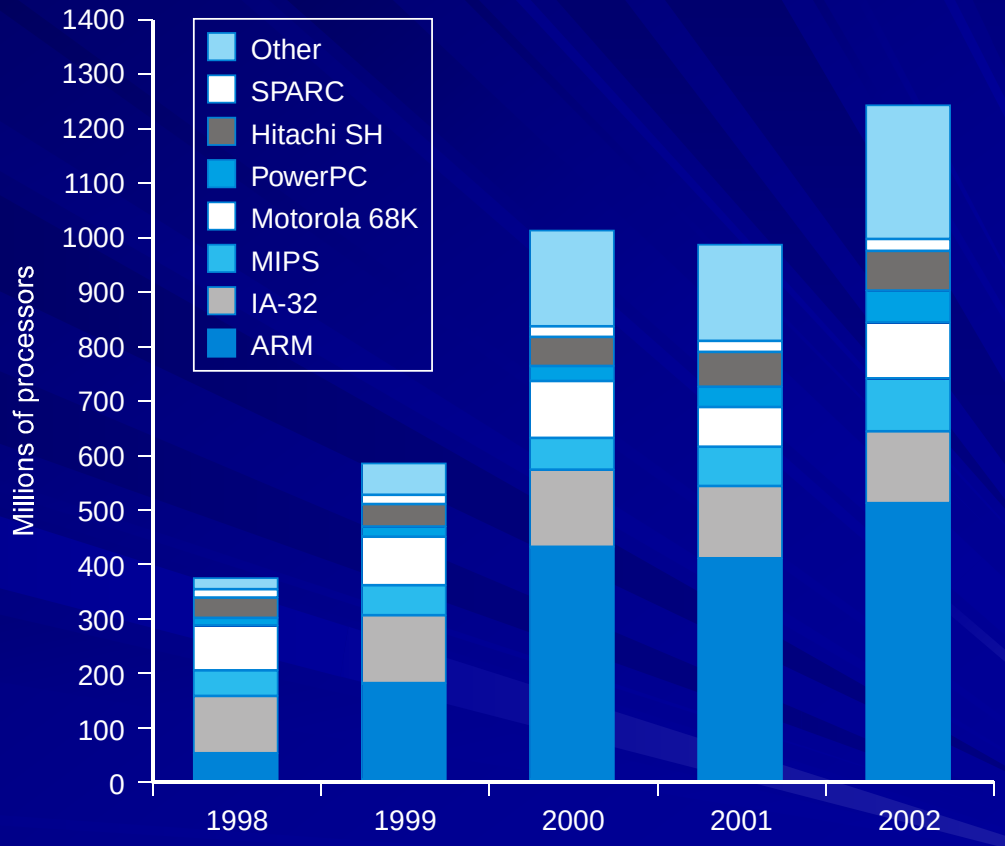
Pipelining of RISC Instructions

Although an instruction takes five clock cycles, one instruction can be completed every cycle.

Instruction execution is overlapped, hardware is utilized more effectively, and instruction throughput is increased. Instruction Level Parallelism (ILP)!

Growth of Processors

- Language of the Machine
- We'll be working with the MIPS instruction set architecture
  - similar to other architectures developed since the 1980's
  - Almost 100 million MIPS processors manufactured in 2002
  - used by NEC, Nintendo, Cisco, Silicon Graphics, Sony, …



MIPS Instruction Set (RISC)

- Instructions execute simple functions.
- Maintain regularity of format – each instruction is one word, contains *opcode* and *arguments*.
- Minimize memory accesses – whenever possible use registers as arguments.
- Three types of instructions:
  - Register (R)-type – only registers as arguments.
  - Immediate (I)-type – arguments are registers and numbers (constants or memory addresses).
  - Jump (J)-type – argument is an address.

MIPS Arithmetic Instructions

- All instructions have 3 operands
- Operand order is fixed (destination first)

Example:

C code: `a = b + c;`

MIPS 'code': `add a, b, c`

"The natural number of operands for an operation like addition is three… requiring every instruction to have exactly three operands conforms to the philosophy of keeping the hardware simple"

Arithmetic Instr. (Continued)

- Design Principle**: simplicity favors regularity**.
- Of course this complicates some things...

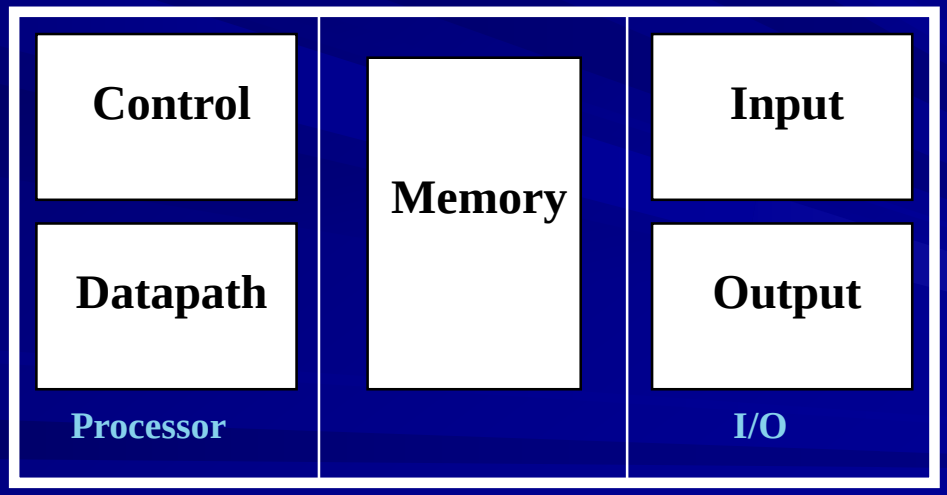
C code: `a = b + c + d;`

MIPS code:
```
add a, b, c
add a, a, d
```

- Operands must be registers (why?)
  - *Remember von Neumann bottleneck.*
- 32 registers provided
- Each register contains 32 bits

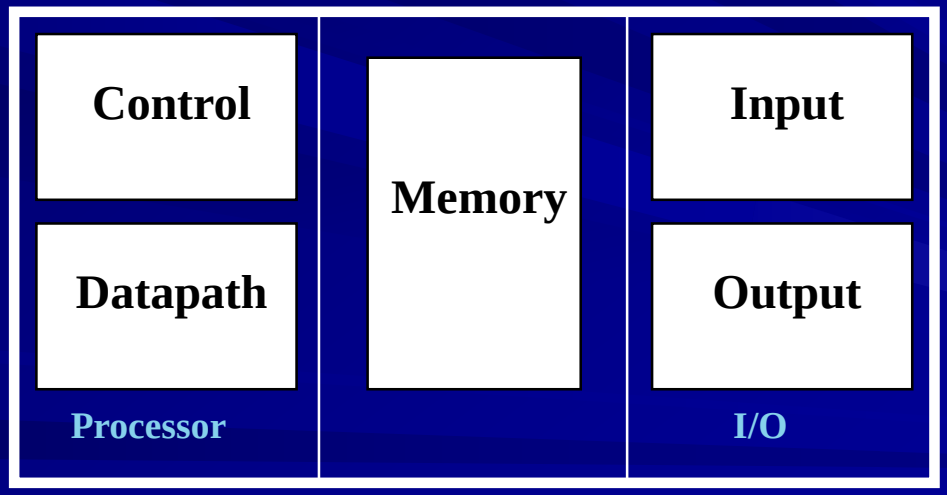
Registers vs. Memory

- Arithmetic instructions operands must be registers
  - **32 registers provided**
- Compiler associates variables with registers.
- What about programs with lots of variables?



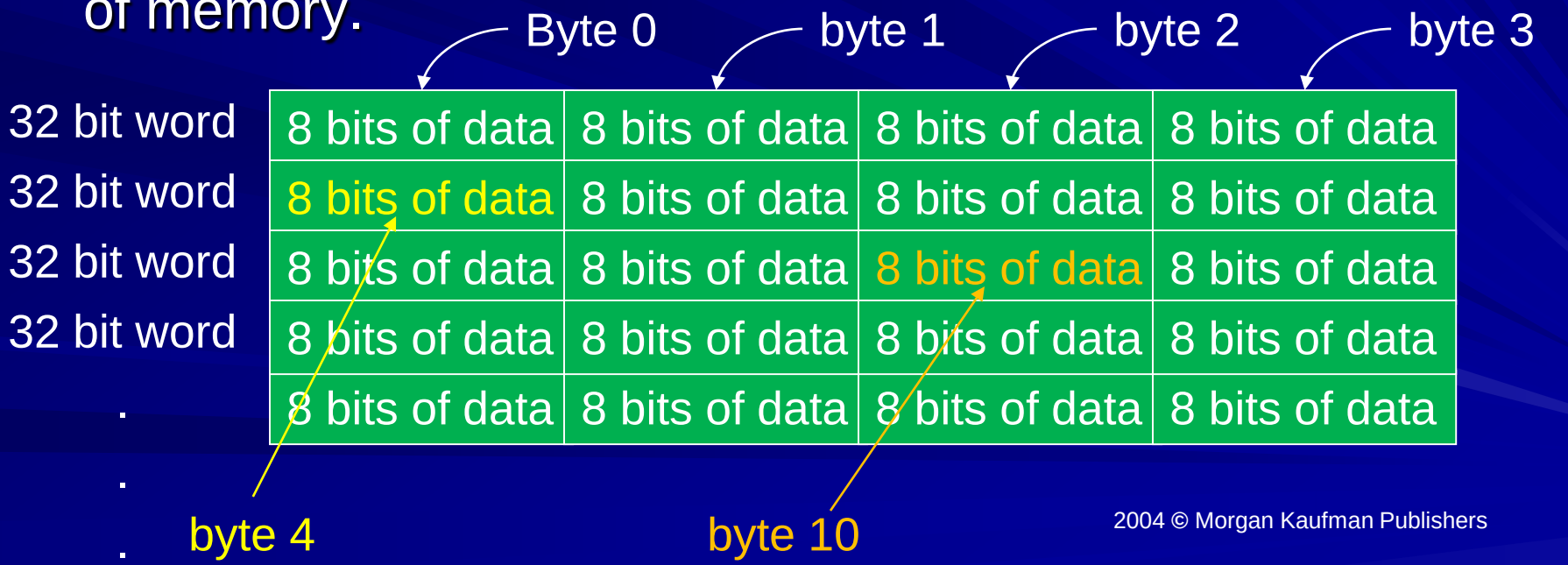
Registers vs. Memory

- Arithmetic instructions operands must be registers
  - **32 registers provided**
- Compiler associates variables with registers.
- What about programs with lots of variables? *Must use memory.*



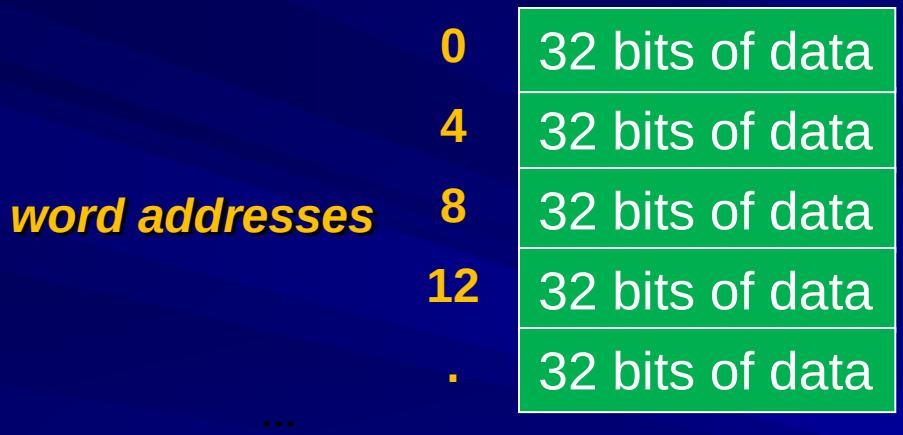
Memory Organization

- Viewed as a large, single-dimension array, with an address.
- A memory address is an index into the array.
- "Byte addressing" means that the index points to a byte of memory.



Memory Organization

- Bytes are nice, but most data items use larger "words"
- For MIPS, a word contains 32 bits or 4 bytes.



Registers hold 32 bits of data

Use 32 bit address

- 2^{32} bytes with addresses from 0 to $2^{32} - 1$
- 2^{30} words with addresses 0, 4, 8, ... $2^{32} - 4$
- Words are aligned

  i.e., what are the least 2 significant bits of a word address?

Instructions

- Load and store instructions
- Example:

C code: `A[12] = h + A[8];`

MIPS code:

```
lw $t0, 32($s3) #addr of A in reg s3
add $t0, $s2, $t0   #h in reg s2
sw $t0, 48($s3)
```

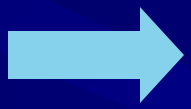
- Can refer to registers by name (e.g., $s2, $t2) instead of number
- Store word has destination last
- Remember arithmetic operands are registers, not memory!

Can't write: `add 48($s3), $s2, 32($s3)`

Our First Example

- Can we figure out the code of subroutine?

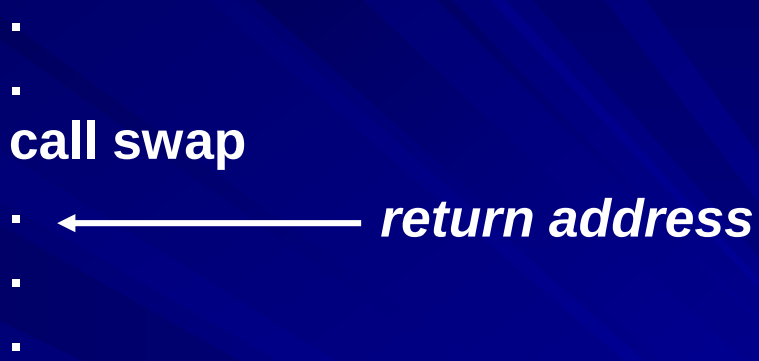
```
swap(int v[], int k);
{ int temp;
   temp = v[k]
   v[k] = v[k+1];
   v[k+1] = temp;
}
```



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

- Initially, k is in reg 5; base address of v is in reg 4; return addr is in reg 31

What Happens?



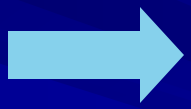
- When the program reaches “call swap” statement:
  - Jump to swap routine
    - Registers 4 and 5 contain the arguments (register convention)
    - Register 31 contains the return address (register convention)
  - Swap two words in memory
  - Jump back to return address to continue rest of the program

Memory and Registers

Our First Example

- Now figure out the code:

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



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

Machine Language

- Instructions, like registers and words of data, are also 32 bits long
  - Example: **`add $t1, $s1, $s2`**
  - registers are numbered, **`$t1=8, $s1=17, $s2=18`**

- Instruction Format:

000000	10001	10010	01000	00000	100000
opcode	**rs**	**rt**	**rd**	**shamt**	**funct**

- *Can you guess what the field names stand for?*

Machine Language

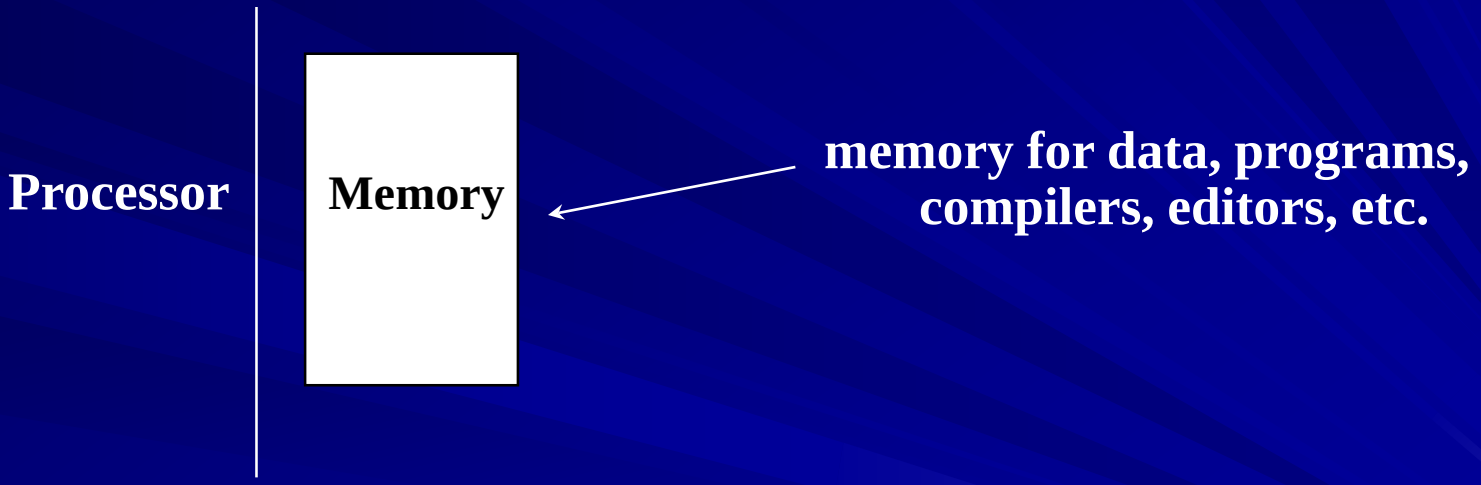
- Consider the load-word and store-word instructions,
  - What would the regularity principle have us do?
  - New principle: Good design demands a compromise
- Introduce a new type of instruction format
  - I-type for data transfer instructions
  - other format was R-type for register
- Example: `lw $t0, 32($s2)`

35	18	9	32
opcode	rs	rt	16 bit number

- Where's the compromise?

Stored Program Concept

- Instructions are bits
- Programs are stored in memory
  - to be read or written just like data



- Fetch and Execute Cycles
  - Instructions are fetched and put into a special register
  - Opcode bits in the register "control" the subsequent actions
  - Fetch the “next” instruction and continue

Control

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

- MIPS conditional branch instructions:

```
bne $t0, $t1, Label
beq $t0, $t1, Label
```

- Example: `if (i==j) h = i + j;`

```
        bne $s0, $s1, Label
        add $s3, $s0, $s1
Label:      ....
```

Control

- MIPS unconditional branch instructions:

```
j  label
```

- Example:

```
if (i!=j)          beq $s4, $s5, Lab1
    h=i+j;         add $s3, $s4, $s5
else               j Lab2
    h=i-j;         Lab1:sub $s3, $s4, $s5
                   Lab2:...
```

- Can you build a simple *for* loop*?*

So Far We've Learned

- Instruction / Meaning

Instruction	Meaning
`add $s1,$s2,$s3`	`$s1 = $s2 + $s3`
`sub $s1,$s2,$s3`	`$s1 = $s2 – $s3`
`lw $s1,100($s2)`	`$s1 = Memory[$s2+100]`
`sw $s1,100($s2)`	`Memory[$s2+100] = $s1`
`bne $s4,$s5,Label`	Next instr. is at Label if $s4 ≠ $s5
`beq $s4,$s5,Label`	Next instr. is at Label if $s4 = $s5
`j Label`	Next instr. is at Label

- Formats:

R	op	rs	rt	rd	shamt	funct
I	op	rs	rt	16 bit address		
J	op	26 bit address				

Three Ways to Jump: j, jr, jal

- j *instr* # jump to machine instruction *instr* (unconditional jump)
- jr $ra # jump to address in register ra (used by calee to go back to caller)
- jal *addr* # set $ra = PC+4 and go to *addr* (jump and link; used to jump to a procedure)

Policy of 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
$t8-$t9	24-25	more temporaries
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address

Register 1 ($at) reserved for assembler, 26-27 for operating system

Control Flow

- We have: beq, bne, what about Branch-if-less-than?
- New instruction:

```
                              if   $s1 < $s2 then
                                   $t0 = 1
slt $t0, $s1, $s2             else
                                   $t0 = 0
```

- Can use this instruction to build new “pseudoinstruction”

```
blt $s1, $s2, Label
```

- Note that the assembler needs a register to do this,
  — there are policy of use conventions for registers

Pseudoinstructions

- blt $s1, $s2, reladdr
- Assembler converts to:

```
slt  $1, $s1, $s2
bne  $1, $zero, reladdr
```

- Other pseudoinstructions: bgt, ble, bge, li, move
- Not implemented in hardware
- Assembler expands pseudoinstructions into machine instructions
- Register 1, called $at, is reserved for converting pseudoinstructions into machine code.

Preview: Project – to be assigned

- Part 1 – Design an instruction set for a 16-bit processor.
  - The ISA may contain no more than 16 unique instructions. However, you may have multiple formats for a given type of instruction, if necessary.
  - Of the 16 instructions, at least one instruction should make your processor **HALT.**
  - The ISA is to support 16-bit data words only. (No byte operands.) All operands are to be 16-bit signed integers (2's complement). Each instruction must be encoded using one 16-bit word.

Project Preview (Cont.)

- The ISA is to support linear addressing of 1K, 16-bit words memory. The memory is to be word-addressable only - **not byte-addressable**.
- The ISA should contain appropriate numbers and types of user-programmable registers to support it. Since this is a small processor, the hardware does not necessarily need to support dedicated registers for stack pointer, frame pointer, etc.
- The ISA must “support” C Programming Language constructs.
- Control flow structures: “if-else” structures, “while” loops, “for” loops.
- Functions (call and return).

Constants

- Small constants are used quite frequently (50% of operands)

  e.g., A = A + 5;
  B = B + 1;
  C = C – 18;

- Solutions? Why not?
  - put 'typical constants' in memory and load them.
  - create hard-wired registers (like $zero) for constants like one.

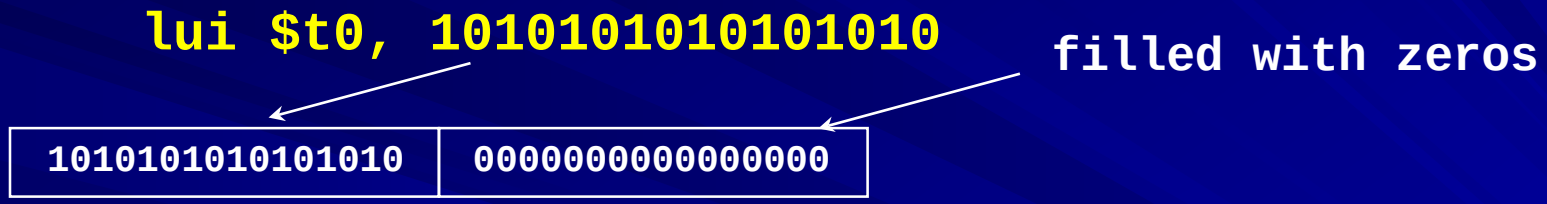
- MIPS Instructions:

```
addi  $29, $29, 4
slti  $8, $18, 10
andi  $29, $29, 6
ori   $29, $29, 4
```

- Design Principle: Make the common case fast. *Which format?*

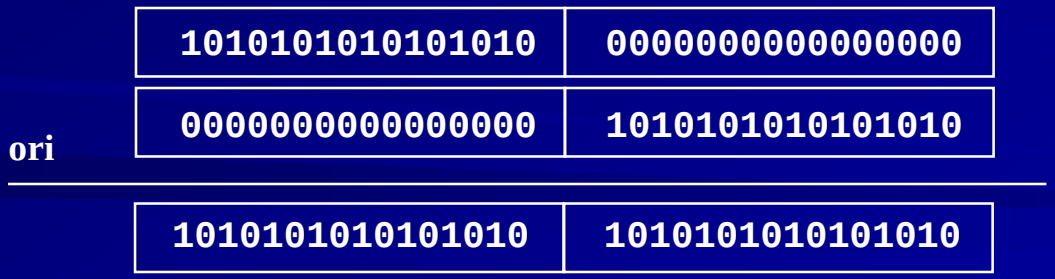
How About Larger Constants?

- We'd like to be able to load a 32 bit constant into a register
- Must use two instructions, new "load upper immediate" instruction



- Then must get the lower order bits right, i.e.,

```
ori $t0, $t0, 1010101010101010
```

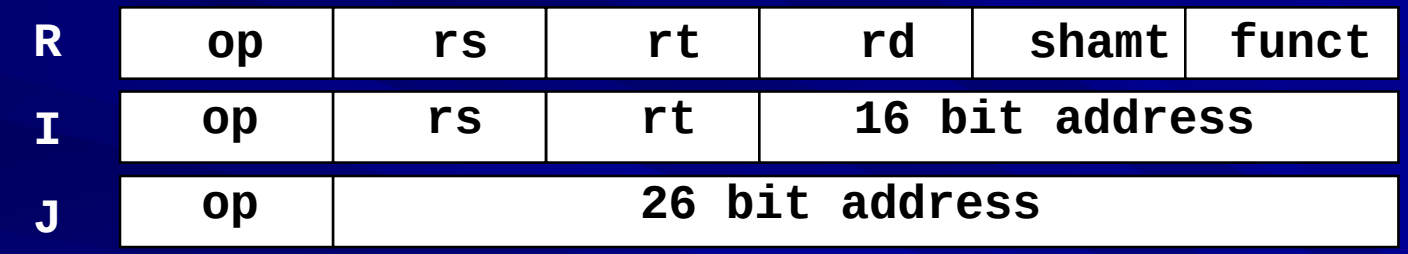


Assembly Language vs. Machine Language

- Assembly provides convenient symbolic representation
  - much easier than writing down numbers
  - e.g., destination first
- Machine language is the underlying reality
  - e.g., destination is no longer first
- Assembly can provide 'pseudoinstructions'
  - e.g., “move $t0, $t1” exists only in Assembly
  - implemented using “add $t0, $t1, $zero”
- When considering performance you should count real instructions and clock cycles

Overview of MIPS

- simple instructions, all 32 bits wide
- very structured, no unnecessary baggage
- only three instruction formats



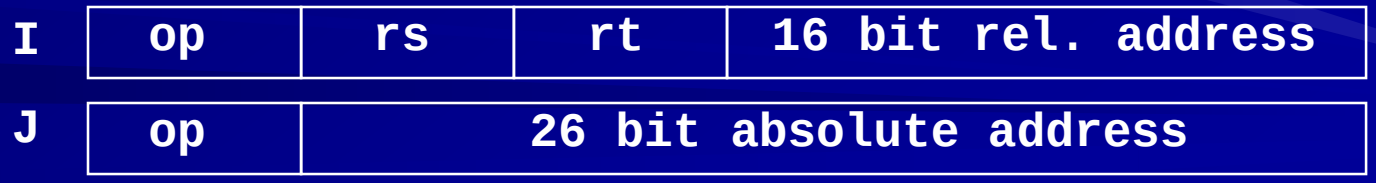
- rely on compiler to achieve performance

Addresses in Branches and Jumps

- Instructions:

`bne $t4, $t5, Label`	Next instruction is at Label if **$t4 ≠ $t5**
`beq $t4, $t5, Label`	Next instruction is at Label if **$t4 = $t5**
`j Label`	Next instruction is at Label

- Formats:



Addresses in Branches

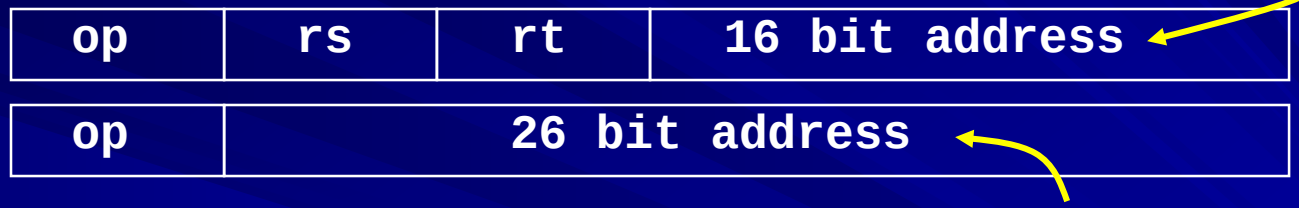
- Instructions:

  `bne $t4,$t5,Label` Next instruction is at Label if $t4 ≠ $t5

  `beq $t4,$t5,Label` Next instruction is at Label if $t4 = $t5

- Formats:

  -2^{15} to $2^{15}-1$ ~ ±32 Kwords



  2^{26} = 64 Mwords

- Relative addressing
  - with respect to PC (program counter)
  - most branches are local (principle of locality)
- Jump instruction just uses high order bits of PC
  - address boundaries of 256 MBytes (maximum jump 64 Mwords)

Example: Loop in C (p. 92)

```
while  ( save[i] == k )
       i += 1;
```

Given a value for k, set i to the index of element in array save [ ] that does not equal k.

MIPS Code for While Loop

Compiler assigns variables to registers:

$s3 (reg 19)	**←**	**i**	**initially 0**
$s5 (reg 21)	**←**	**k**	
$s6 (reg 22)	**←**	**memory address where save [ ] begins**	

Then generates the following assembly code:

```
Loop: sll   $t1, $s3, 2      # Temp reg $t1 = 4 * i
      add   $t1, $t1, $s6    # $t1 = address of save[i]
      lw    $t0, 0($t1)      # Temp reg $t0 = save[i]
      bne   $t0, $s5, Exit   # go to Exit if save[i] ≠ k
      addi  $s3, $s3, 1      # i = i + 1
      j     Loop             # go to Loop
Exit:
```

Machine Code and Mem. Adresses

Memory Byte addr.	*Machine code* Bits 31-26	25-21	20-16	15-11	10 – 6	5 – 0	
80000	0	0	19	9	2	0	sll
80004	0	9	22	9	0	32	add
80008	35	9	8	0			lw
80012	5	8	21	Exit = +2			bne
80016	8	19	19	1			addi
80020	2	Loop = 20000 (memory word address)					j
80024	. . . . .						

Note: $t0 ≡ Reg 8, $t1 ≡ Reg 9, $s3 ≡ Reg 19, $s5 ≡ Reg 21, $s6 ≡ Reg 22
temp temp i k save

Finding Branch Address *Exit*

- Exit = +2 is a 16 bit integer in bne instruction

  000101 01000 10101 0000000000000010 = 2
- $PC = 80016 is the byte address of the next instruction

  00000000000000010011100010010000 = 80016
- Multiply bne argument by 4 (convert to byte address)

  0000000000001000 = 8
- $PC ← $PC + 8

  00000000000000010011100010011000 = 80024

  Thus, *Exit* is memory byte address 80024.

Finding Jump Address *Loop*

- J 20000

  000010 00000000000100111000100000 = 20000
- $PC = 80024, when jump is being executed

  0000000000000100111000010011000 = 80024
- Multiply J argument by 4 (convert to byte address)

  00000000000100111000010000000 = 80000
- Insert four leading bits from $PC

  0000000000000100111000010000000 = 80000

  Thus, *Loop* is memory byte address 80000.

Summary: MIPS Registers and Memory

	`$s0-$s7, $t0-$t9, $zero,`	Fast locations for data. In MIPS, data must be in registers to perform
32 registers	`$a0-$a3, $v0-$v1, $gp,`	arithmetic. MIPS register $zero always equals 0. Register $at is
	`$fp, $sp, $ra, $at`	reserved for the assembler to handle large constants.
	Memory[0],	Accessed only by data transfer instructions. MIPS uses byte
2^{30} memory	Memory[4], ...,	addresses, so sequential words differ by 4. Memory holds data
words	Memory[4294967292]	structures, such as arrays, and spilled registers, such as those
		saved on procedure calls.

Summary: MIPS Instructions

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	branch on not equal	`bne $s1, $s2, 25`	if (`$s1 != $s2`) go to PC + 4 + 100	Not equal test; PC-relative
	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	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

Addressing Modes

Example

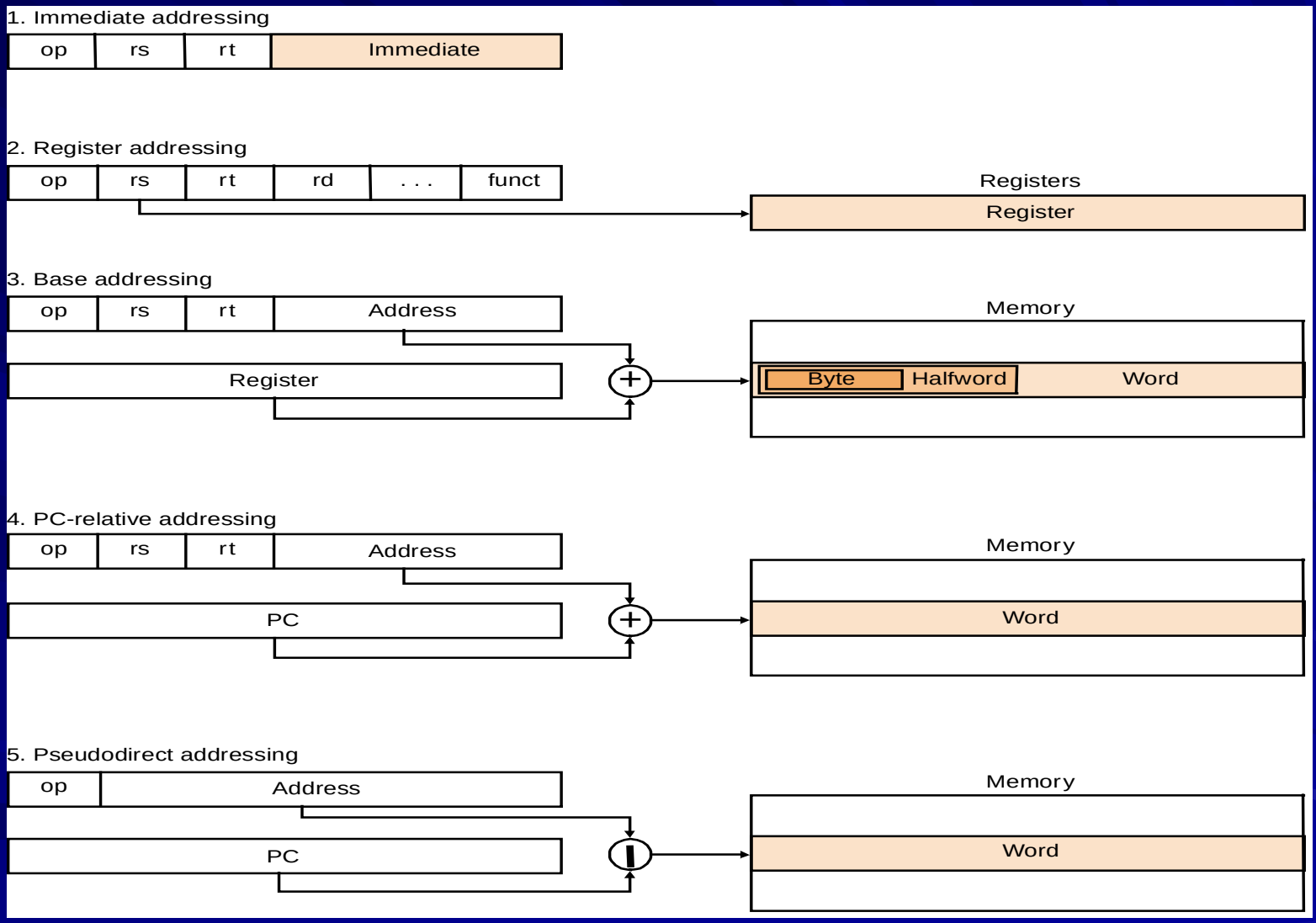
addi

add

lw, sw

beq, bne

j



Alternative Architectures

- Design alternative:
  - provide more powerful operations
  - goal is to reduce number of instructions executed
  - danger is a slower cycle time and/or a higher CPI

    –***"The path toward operation complexity is thus fraught with peril. To avoid these problems, designers have moved toward simpler instructions"***

- Let's look (briefly) at IA-32 (Sec 2.17 in text)

IA–32 (a.k.a. x86)

- 1978: The Intel 8086 is announced (16 bit architecture)
- 1980: The 8087 floating point coprocessor is added
- 1982: The 80286 increases address space to 24 bits, +instructions
- 1985: The 80386 extends to 32 bits, new addressing modes
- 1989-1995: The 80486, Pentium, Pentium Pro add a few instructions (mostly designed for higher performance)
- 1997: 57 new “MMX” instructions are added, Pentium II
- 1999: The Pentium III added another 70 instructions (SSE – streaming SIMD extensions)
- 2001: Another 144 instructions (SSE2)
- 2003: AMD extends the architecture to increase address space to 64 bits, widens all registers to 64 bits and makes other changes (AMD64)
- 2004: Intel capitulates and embraces AMD64 (calls it EM64T) and adds more media extensions
- *“This history illustrates the impact of the “golden handcuffs” of compatibility:*
  “adding new features as someone might add clothing to a packed bag”
  “an architecture that is difficult to explain and impossible to love”

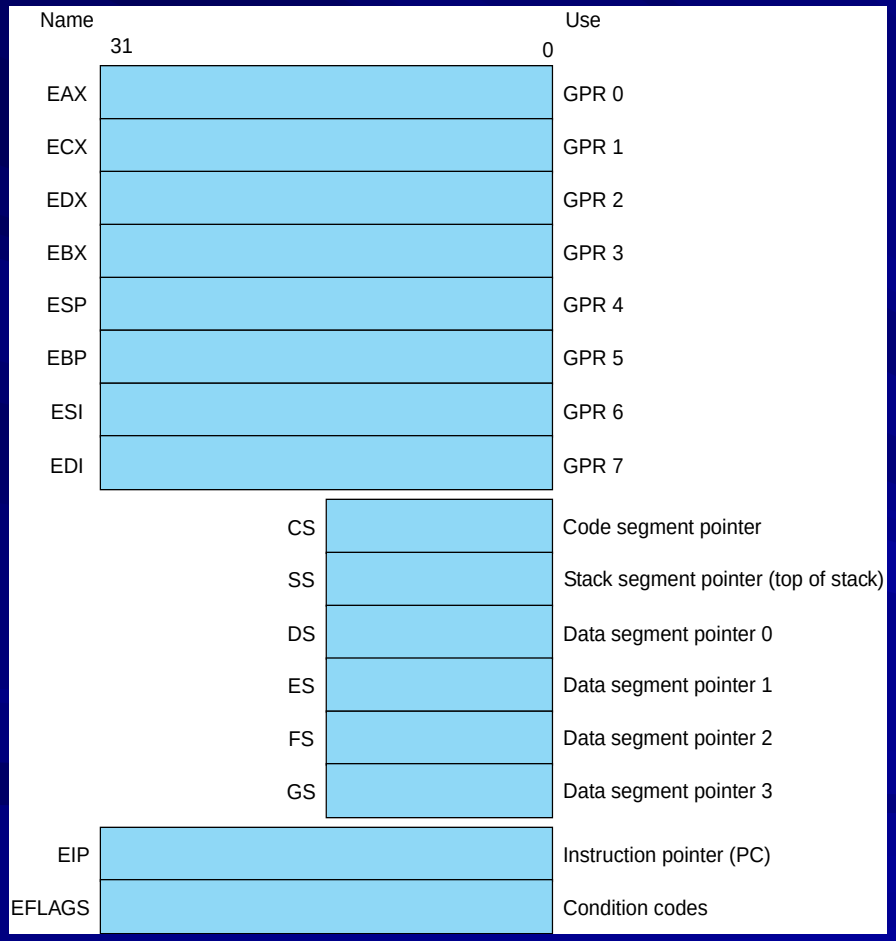
IA-32 Overview

- Complexity:
  - Instructions from 1 to 17 bytes long
  - one operand must act as both a source and destination
  - one operand can come from memory
  - complex addressing modes
    e.g., “base or scaled index with 8 or 32 bit displacement”
- Saving grace:
  - the most frequently used instructions are not too difficult to build
  - compilers avoid the portions of the architecture that are slow

“what the x86 lacks in style is made up in quantity, making it beautiful from the right perspective”

IA-32 Registers

- Registers in the 32-bit subset that originated with 80386



Eight general purpose registers

IA-32 Register Restrictions

- Fourteen major registers.
- Eight 32-bit general purpose registers.
  - ESP or EBP cannot contain memory address.
  - ESP cannot contain displacement from base address.
  - . . .
- See Figure 2.36, page 153 of textbook.

IA-32 Typical Instructions

- Four major types of integer instructions:
  - Data movement including move, push, pop
  - Arithmetic and logical (destination register or memory)
  - Control flow (use of condition codes / flags)
  - String instructions, including string move and string compare

Instruction	Function
JE name	if equal(condition code) {EIP=name}; EIP-128 ≤ name < EIP+128
JMP name	EIP=name
CALL name	SP=SP-4; M[SP]=EIP+5; EIP=name;
MOVW EBX,[EDI+45]	EBX=M[EDI+45]
PUSH ESI	SP=SP-4; M[SP]=ESI
POP EDI	EDI=M[SP]; SP=SP+4
ADD EAX,#6765	EAX= EAX+6765
TEST EDX,#42	Set condition code (flags) with EDX and 42
MOVSL	M[EDI]=M[ESI]; EDI=EDI+4; ESI=ESI+4

FIGURE 2.43 Some typical IA-32 instructions and their functions. A list of frequent operations appears in Figure 2.44. The CALL saves the EIP of the next instruction on the stack. (EIP is the Intel PC.)

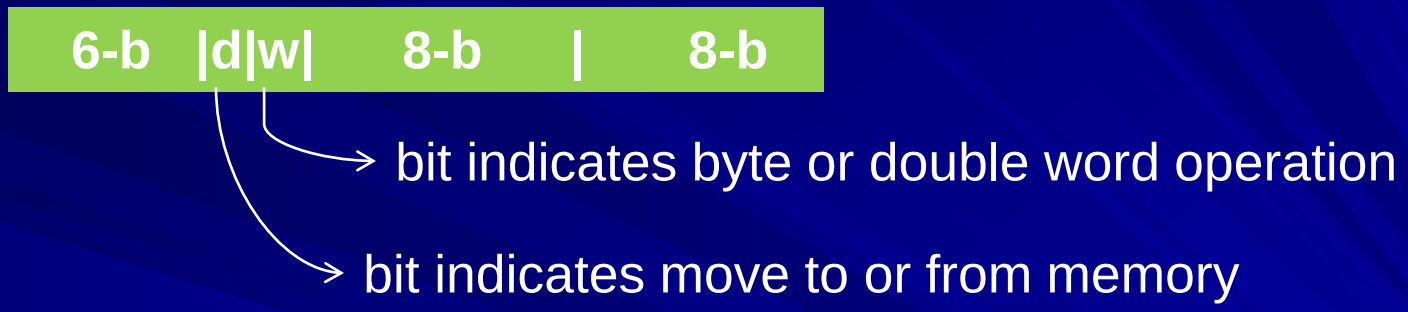
Some IA-32 Instructions

- PUSH 5-bit opcode, 3-bit register operand

- JE 4-bit opcode, 4-bit condition, 8-bit jump offset

Some IA-32 Instructions

- MOV — 6-bit opcode, 8-bit register/mode*, 8-bit offset



- XOR — 8-bit opcode, 8-bit reg/mode*, 8-bit base, 8-b index

*8-bit register/mode: See Figure 2.42, page 158 of textbook.

Some IA-32 Instructions

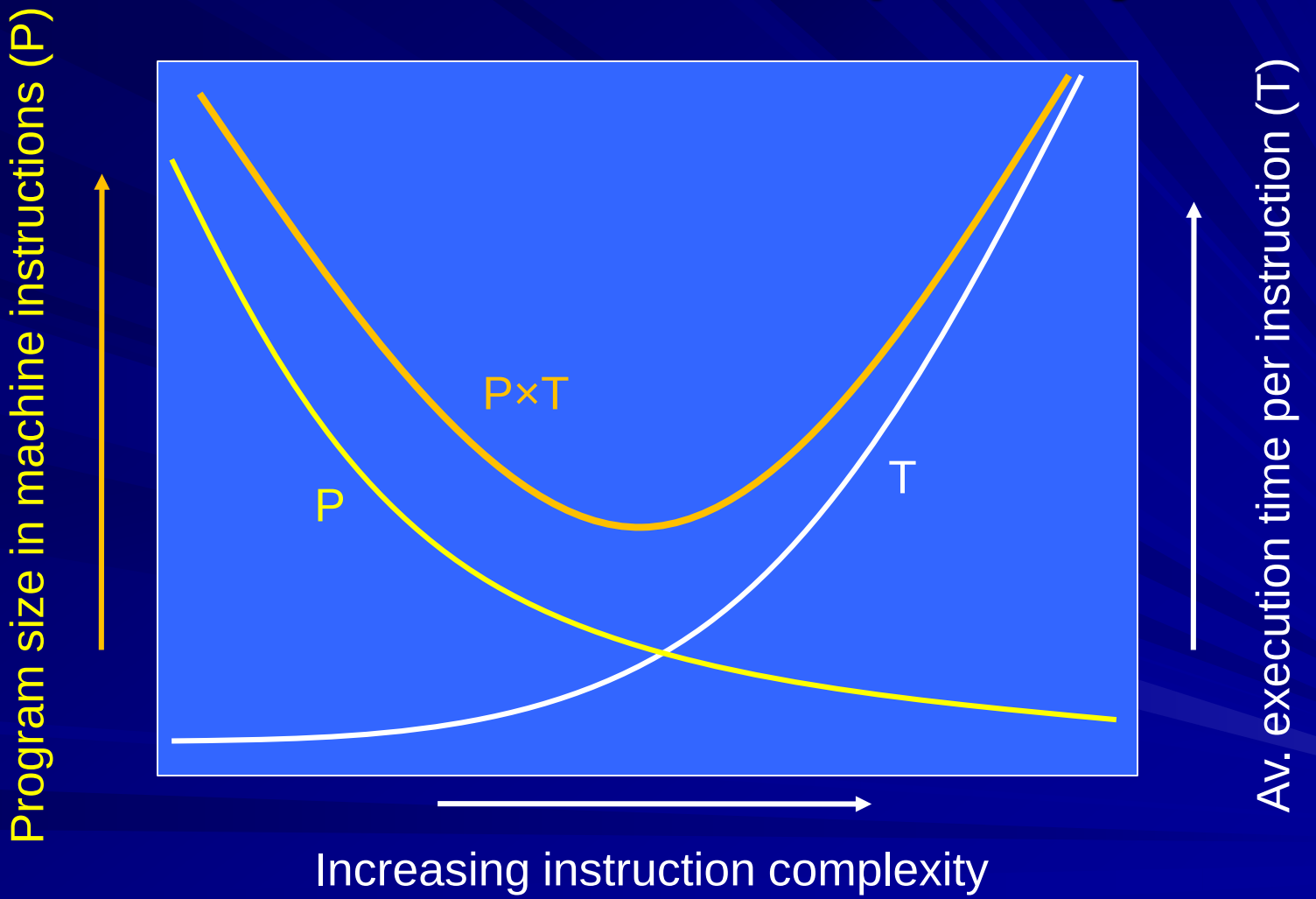
- ADD 4-bit opcode, 3-bit register, 32-bit immediate

- TEST 7-bit opcode, 8-bit reg/mode, 32-bit immediate

Additional References

- IA-32, IA-64 (CISC)
  - A. S. Tanenbaum, *Structured Computer Organization, Fifth Edition*, Upper Saddle River, New Jersey: Pearson Prentice-Hall, 2006, Chapter 5.
- ARM (RISC)
  - D. Seal, *ARM Architecture Reference Manual, Second Edition*, Addison-Wesley Professional, 2000.
- SPARC (Scalable Processor Architecture)
- PowerPC
  - V. C. Hamacher, Z. G. Vranesic and S. G. Zaky, Computer Organization, Fourth Edition, New York: McGraw-Hill, 1996.

Instruction Complexity



URISC: The Other Extreme

- Instruction set has a single instruction:

  label: urisc dest, src1, target

Subtract operand 1 from operand 2, replace operand 2 with the result, and jump to target address if the result is negative.

- See, B. Parhami, *Computer Architecture, from Microprocessors to Supercomputers*, New York: Oxford, 2005, pp. 151-153.

Summary

- Instruction complexity is only one variable
  - lower instruction count vs. higher CPI / lower clock rate – *we will see performance measures later*
- Design Principles:
  - simplicity favors regularity
  - smaller is faster
  - good design demands compromise
  - make the common case fast
- Instruction set architecture
  - a very important abstraction indeed!
- Links to some instruction sets – next slide.

Alternative Instruction Sets

- MIPS

  http://www.d.umn.edu/~gshute/mips/MIPS.html
- ARM

  http://simplemachines.it/doc/arm_inst.pdf
- IA32/64

  http://brokenthorn.com/Resources/OSDevX86.html
- PowerPC

  http://pds.twi.tudelft.nl/vakken/in101/labcourse/instruction-set/
- SPARC

  http://www.cs.unm.edu/~maccabe/classes/341/labman/node9