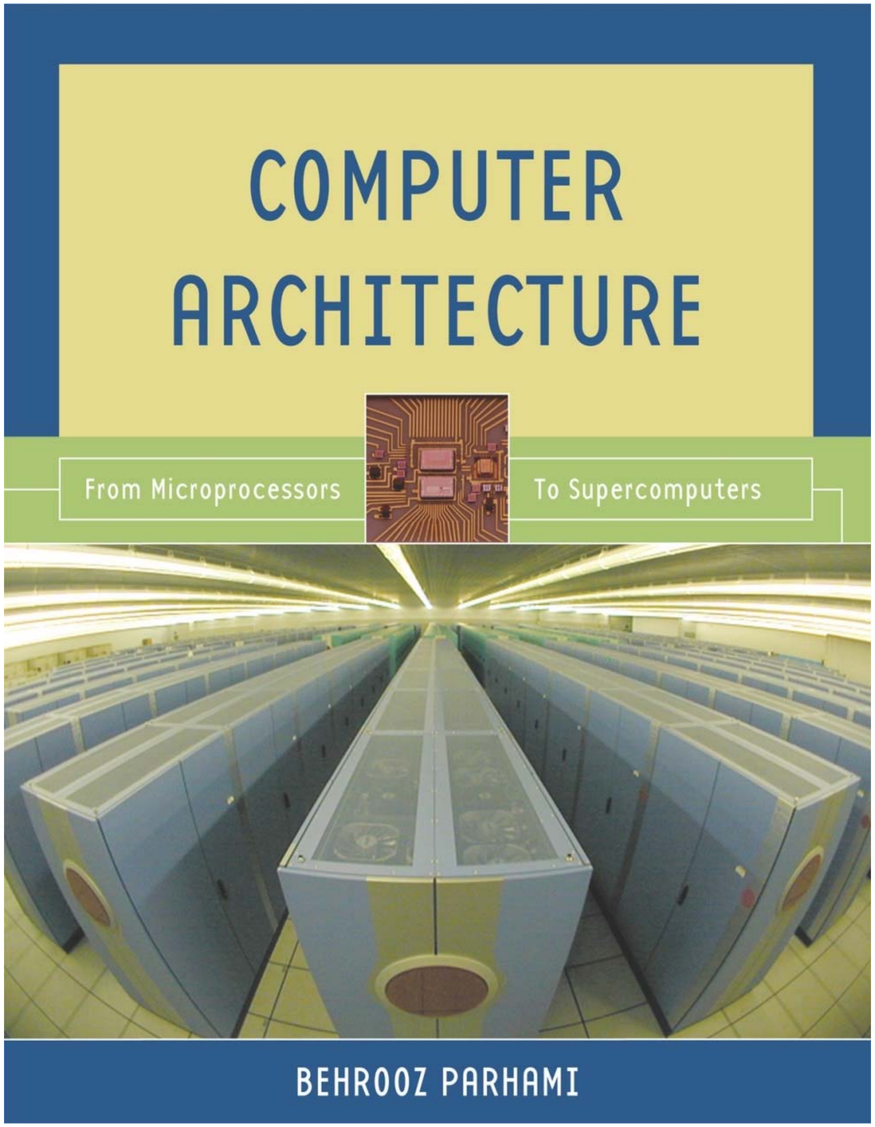
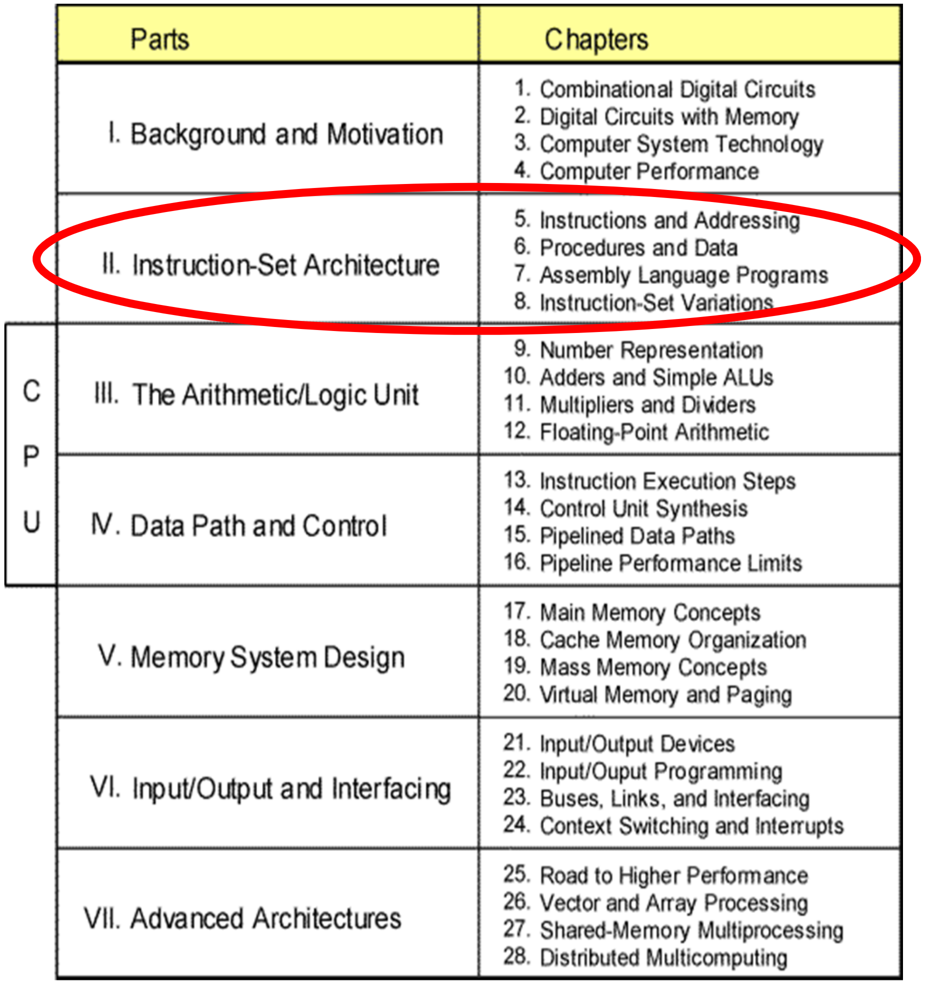


Part II
Instruction-Set Architecture



	Parts	Chapters
	I. Background and Motivation	1. Combinational Digital Circuits 2. Digital Circuits with Memory 3. Computer System Technology 4. Computer Performance
	II. Instruction-Set Architecture	5. Instructions and Addressing 6. Procedures and Data 7. Assembly Language Programs 8. Instruction-Set Variations
CPU	III. The Arithmetic/Logic Unit	9. Number Representation 10. Adders and Simple ALUs 11. Multipliers and Dividers 12. Floating-Point Arithmetic
	IV. Data Path and Control	13. Instruction Execution Steps 14. Control Unit Synthesis 15. Pipelined Data Paths 16. Pipeline Performance Limits
	V. Memory System Design	17. Main Memory Concepts 18. Cache Memory Organization 19. Mass Memory Concepts 20. Virtual Memory and Paging
	VI. Input/Output and Interfacing	21. Input/Output Devices 22. Input/Ouput Programming 23. Buses, Links, and Interfacing 24. Context Switching and Interrupts
	VII. Advanced Architectures	25. Road to Higher Performance 26. Vector and Array Processing 27. Shared-Memory Multiprocessing 28. Distributed Multicomputing

About This Presentation

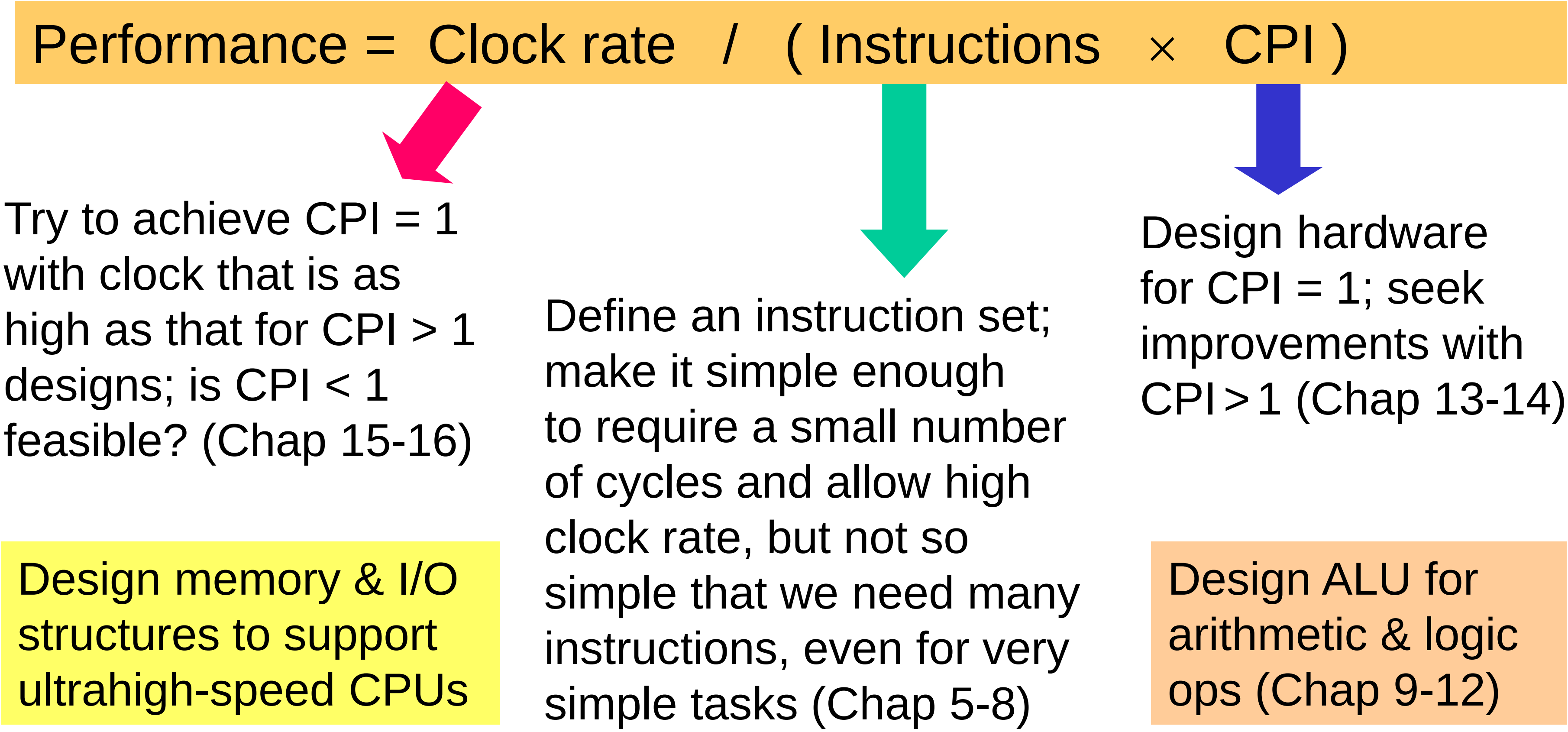
This presentation is intended to support the use of the textbook *Computer Architecture: From Microprocessors to Supercomputers*, Oxford University Press, 2005, ISBN 0-19-515455-X. It is updated regularly by the author as part of his teaching of the upper-division course ECE 154, Introduction to Computer Architecture, at the University of California, Santa Barbara. Instructors can use these slides freely in classroom teaching and for other educational purposes. Any other use is strictly prohibited. © Behrooz Parhami

Edition	Released	Revised	Revised	Revised	Revised
First	June 2003	July 2004	June 2005	Mar. 2006	Jan. 2007
		Jan. 2008	Jan. 2009	Jan. 2011	Oct. 2014

A Few Words About Where We Are Headed

Performance = 1 / Execution time   simplified to   1 / CPU execution time

CPU execution time = Instructions × CPI / (Clock rate)



Performance = Clock rate / (Instructions × CPI)

Try to achieve CPI = 1 with clock that is as high as that for CPI > 1 designs; is CPI < 1 feasible? (Chap 15-16)

Define an instruction set; make it simple enough to require a small number of cycles and allow high clock rate, but not so simple that we need many instructions, even for very simple tasks (Chap 5-8)

Design hardware for CPI = 1; seek improvements with CPI > 1 (Chap 13-14)

Design memory & I/O structures to support ultrahigh-speed CPUs

Design ALU for arithmetic & logic ops (Chap 9-12)

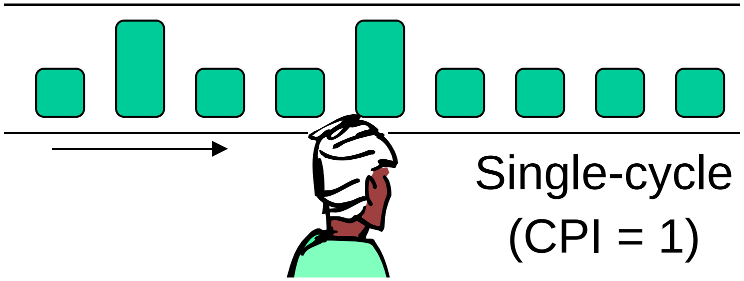
Strategies for Speeding Up Instruction Execution

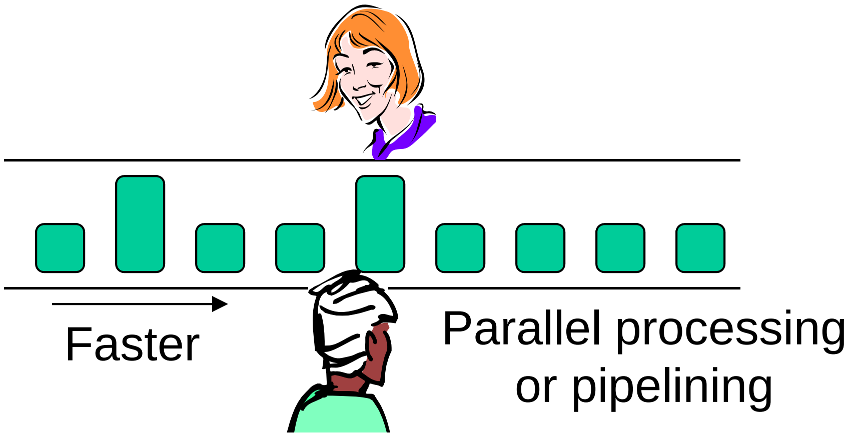
Performance = 1 / Execution time   simplified to   1 / CPU execution time

CPU execution time = Instructions × CPI / (Clock rate)

Performance = Clock rate / (Instructions × CPI)

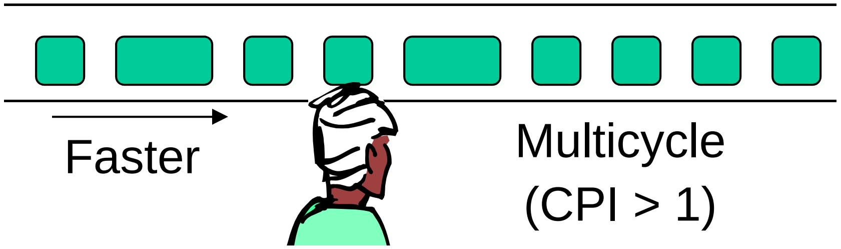
Assembly line analogy


Single-cycle
(CPI = 1)


Faster

Parallel processing
or pipelining

Items that take longest to inspect dictate the speed of the assembly line


Faster

Multicycle
(CPI > 1)

II Instruction Set Architecture

Introduce machine “words” and its “vocabulary,” learning:

- A simple, yet realistic and useful instruction set
- Machine language programs; how they are executed
- RISC vs CISC instruction-set design philosophy

Topics in This Part	
Chapter 5	Instructions and Addressing
Chapter 6	Procedures and Data
Chapter 7	Assembly Language Programs
Chapter 8	Instruction Set Variations

5 Instructions and Addressing

First of two chapters on the instruction set of MiniMIPS:

- Required for hardware concepts in later chapters
- Not aiming for proficiency in assembler programming

Topics in This Chapter
5.1 Abstract View of Hardware
5.2 Instruction Formats
5.3 Simple Arithmetic / Logic Instructions
5.4 Load and Store Instructions
5.5 Jump and Branch Instructions
5.6 Addressing Modes

5.1 Abstract View of Hardware

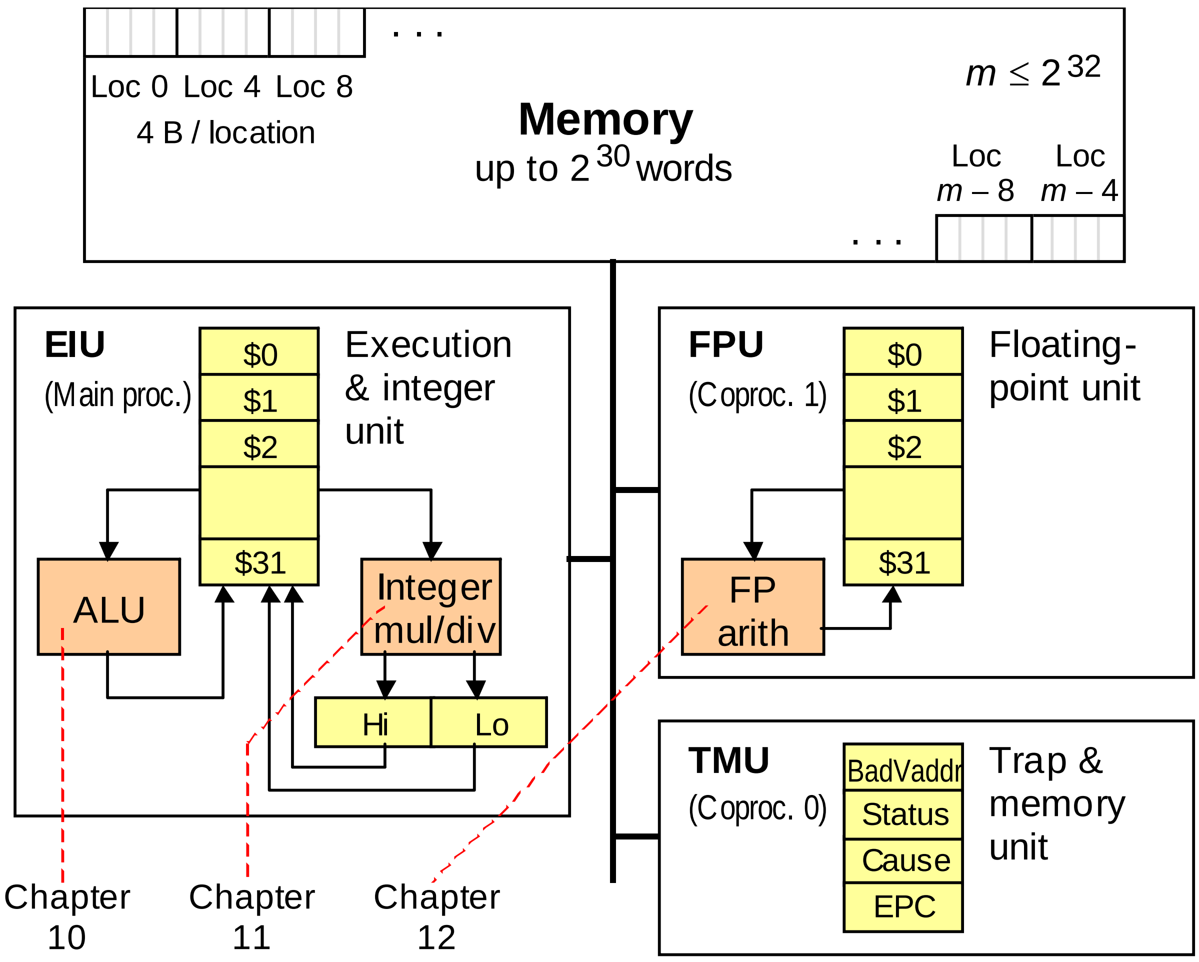
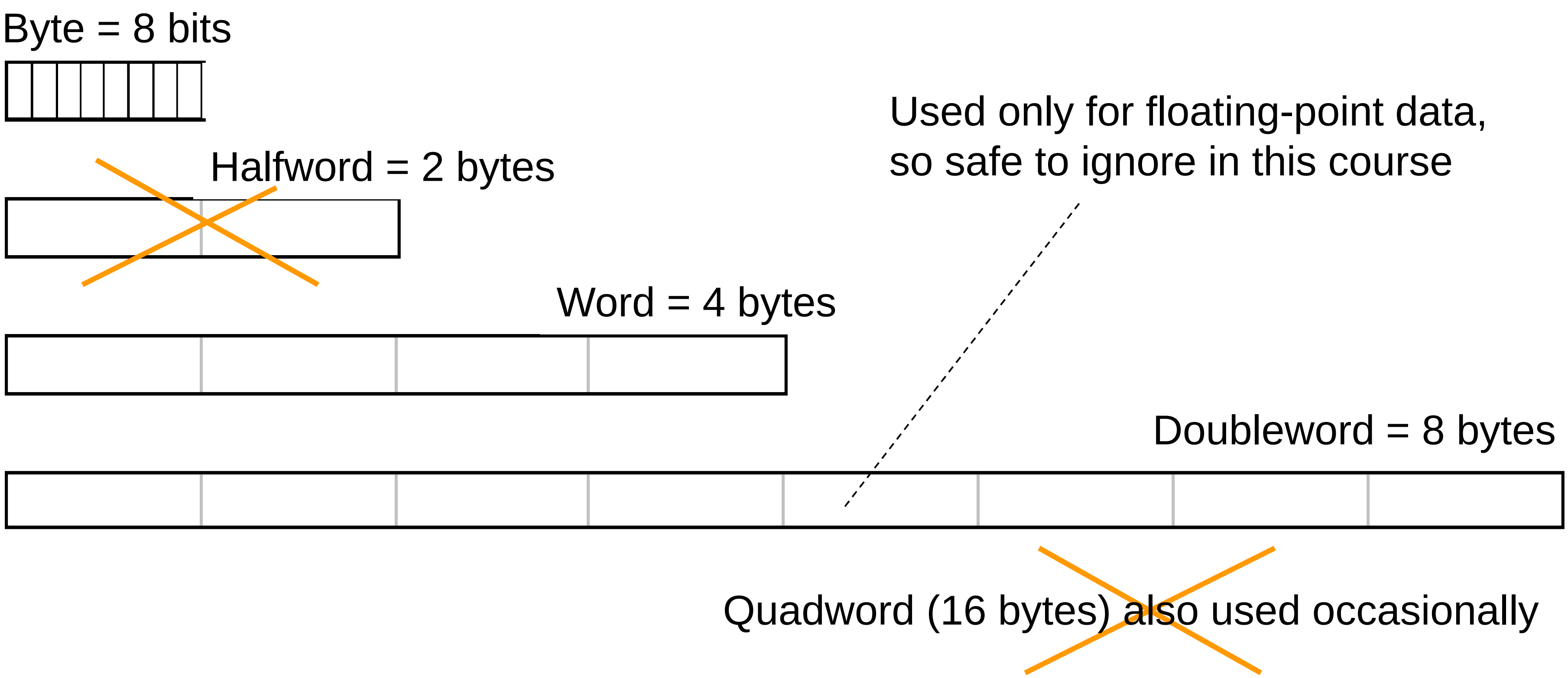


Figure 5.1 Memory and processing subsystems for MiniMIPS.

Data Types


Byte = 8 bits

Halfword = 2 bytes

Word = 4 bytes

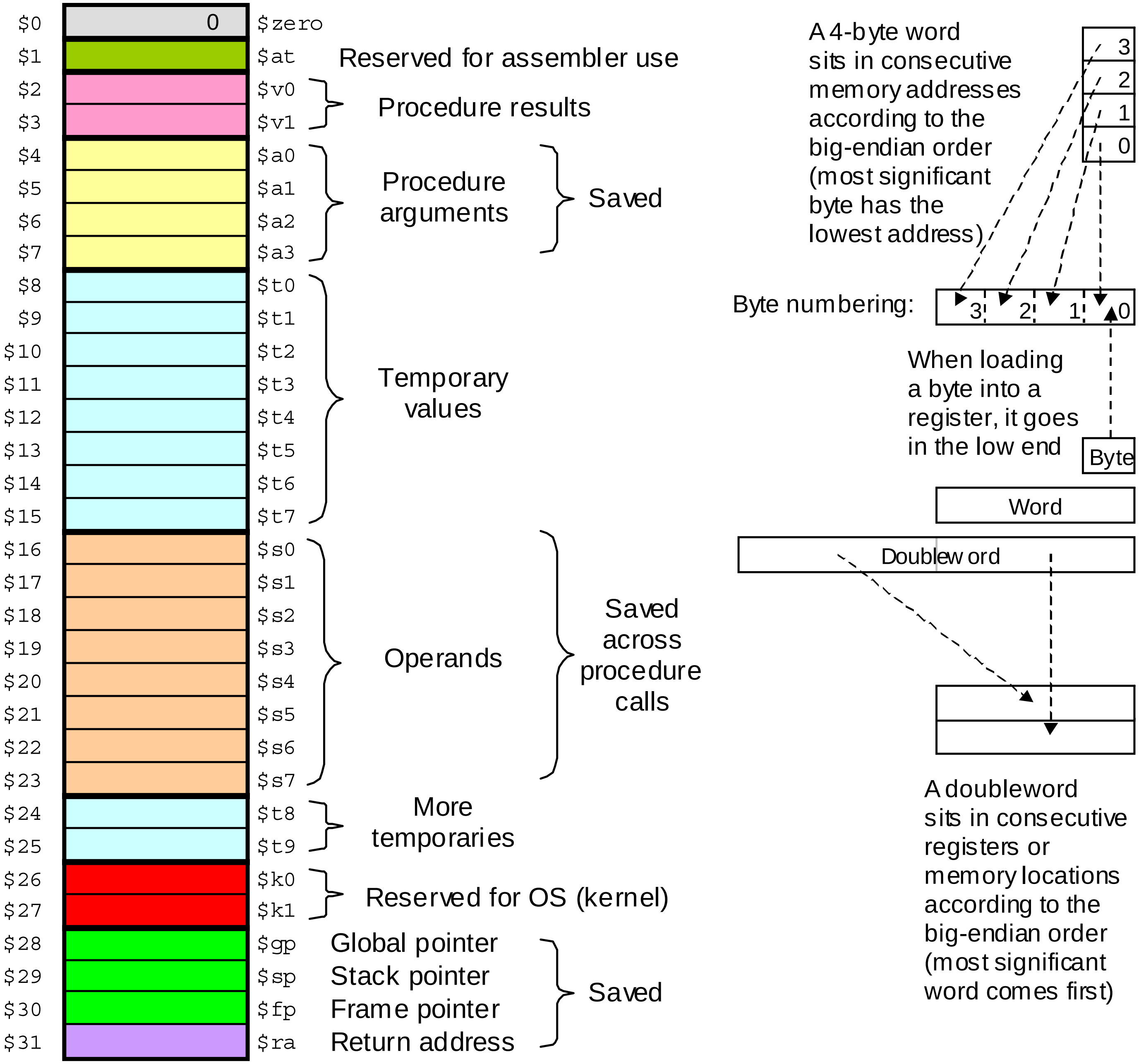
Doubleword = 8 bytes

Used only for floating-point data, so safe to ignore in this course

Quadword (16 bytes) also used occasionally

MiniMIPS registers hold 32-bit (4-byte) words. Other common data sizes include byte, halfword, and doubleword.

Register Conventions



$0	0	$zero
$1		$at
$2		$v0
$3		$v1
$4		$a0
$5		$a1
$6		$a2
$7		$a3
$8		$t0
$9		$t1
$10		$t2
$11		$t3
$12		$t4
$13		$t5
$14		$t6
$15		$t7
$16		$s0
$17		$s1
$18		$s2
$19		$s3
$20		$s4
$21		$s5
$22		$s6
$23		$s7
$24		$t8
$25		$t9
$26		$k0
$27		$k1
$28		$gp
$29		$sp
$30		$fp
$31		$ra

Reserved for assembler use

Procedure results

Procedure arguments

Saved

Temporary values

Operands

Saved across procedure calls

More temporaries

Reserved for OS (kernel)

Global pointer

Stack pointer

Frame pointer

Return address

Saved

A 4-byte word sits in consecutive memory addresses according to the big-endian order (most significant byte has the lowest address)

Byte numbering:

When loading a byte into a register, it goes in the low end

A doubleword sits in consecutive registers or memory locations according to the big-endian order (most significant word comes first)

Figure 5.2 Registers and data sizes in MiniMIPS.

Registers Used in This Chapter

10 temporary registers

8 operand registers

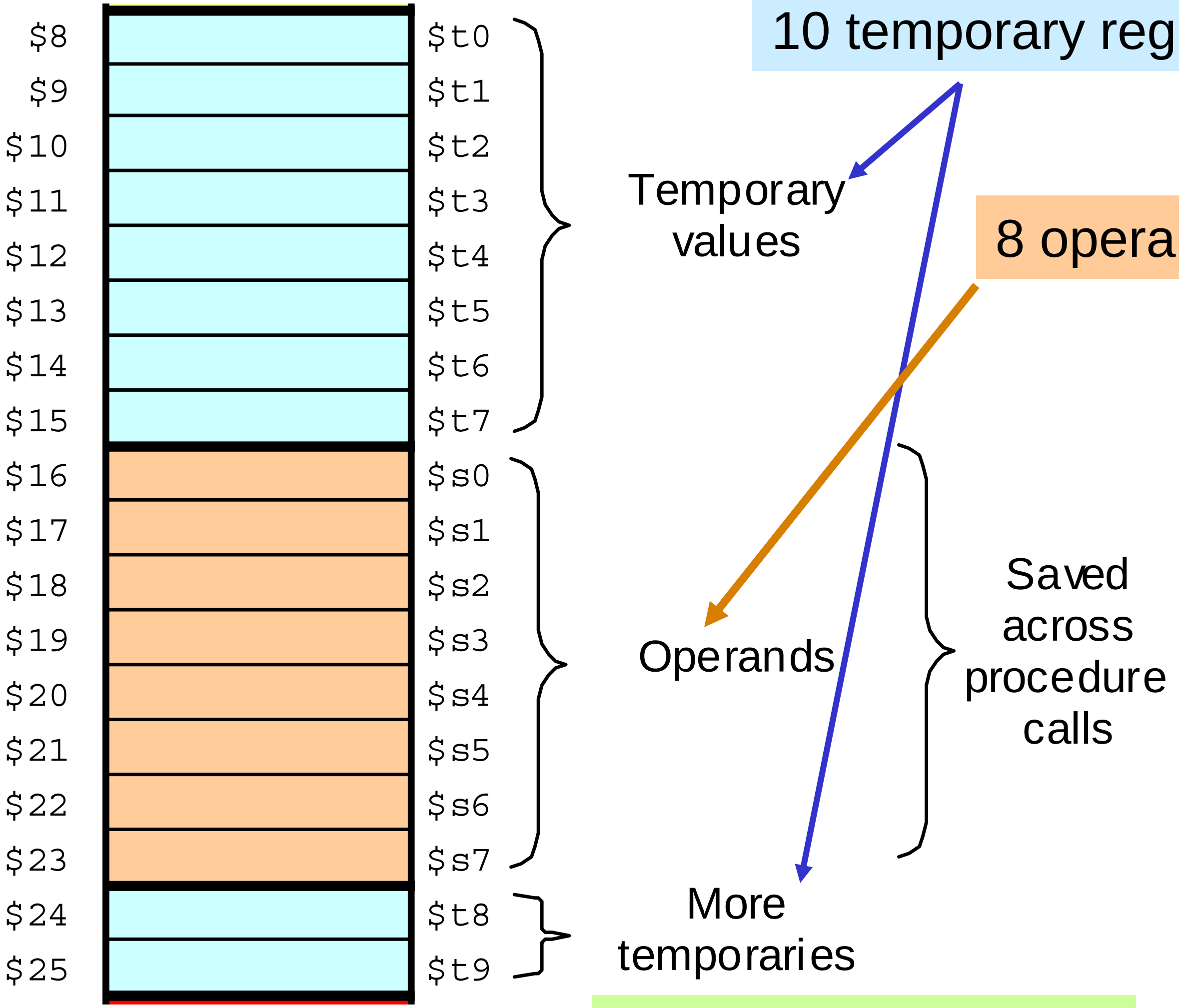
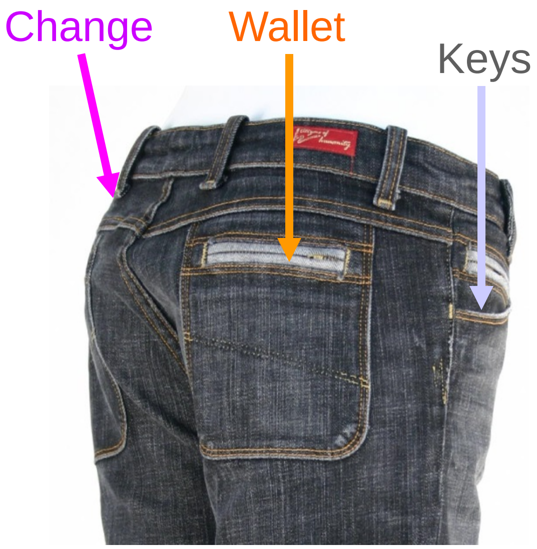
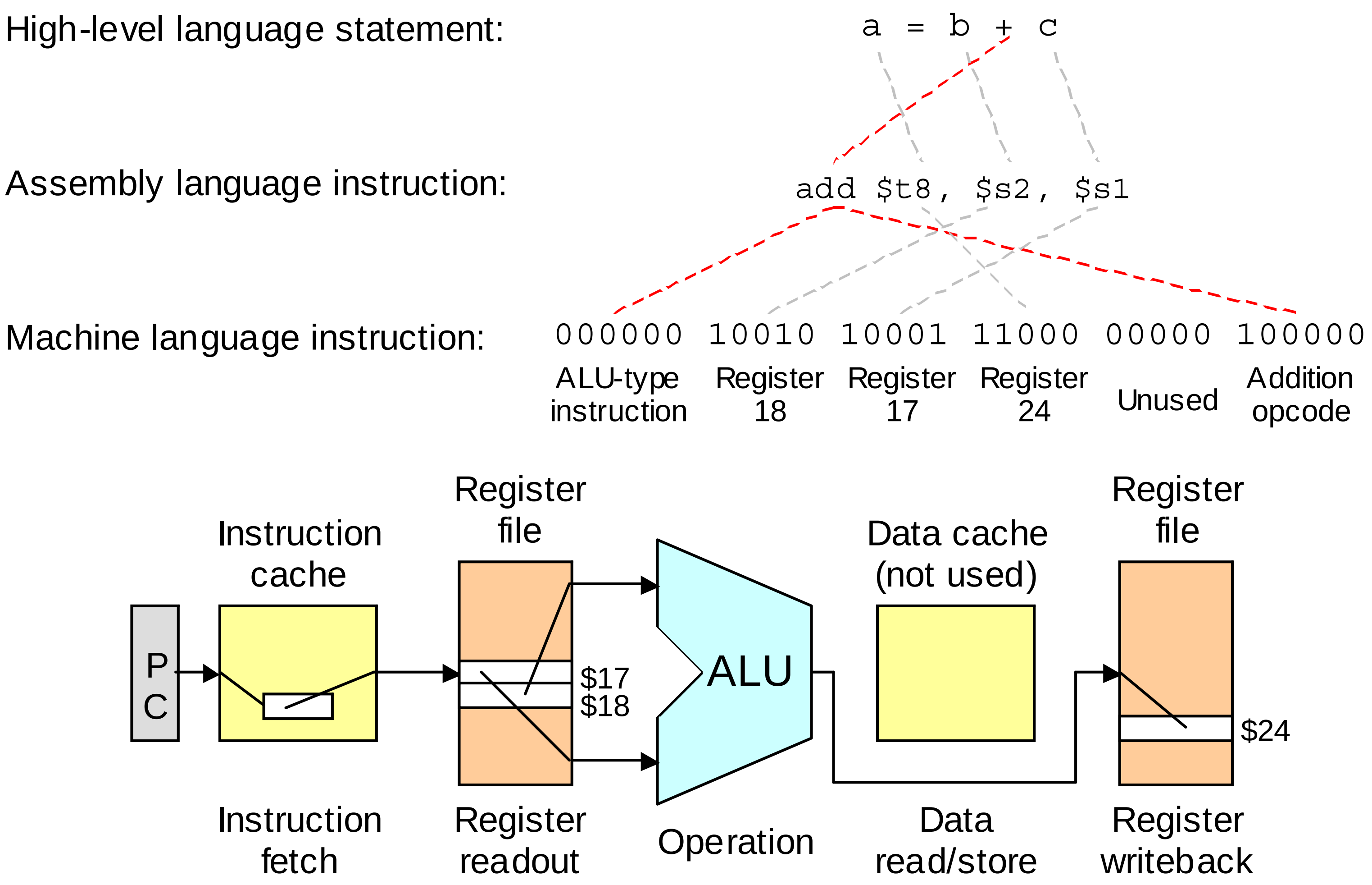


Figure 5.2 (partial)



Analogy for register usage conventions

5.2 Instruction Formats



High-level language statement: `a = b + c`

Assembly language instruction: `add $t8, $s2, $s1`

Machine language instruction:

000000	10010	10001	11000	00000	100000
ALU-type instruction	Register 18	Register 17	Register 24	Unused	Addition opcode

Figure 5.3   A typical instruction for MiniMIPS and steps in its execution.

Add, Subtract, and Specification of Constants

MiniMIPS add & subtract instructions; e.g., compute:

```
g = (b + c) – (e + f)
```

```
add  $t8,$s2,$s3     # put the sum b + c in $t8
add  $t9,$s5,$s6     # put the sum e + f in $t9
sub  $s7,$t8,$t9     # set g to ($t8) – ($t9)
```

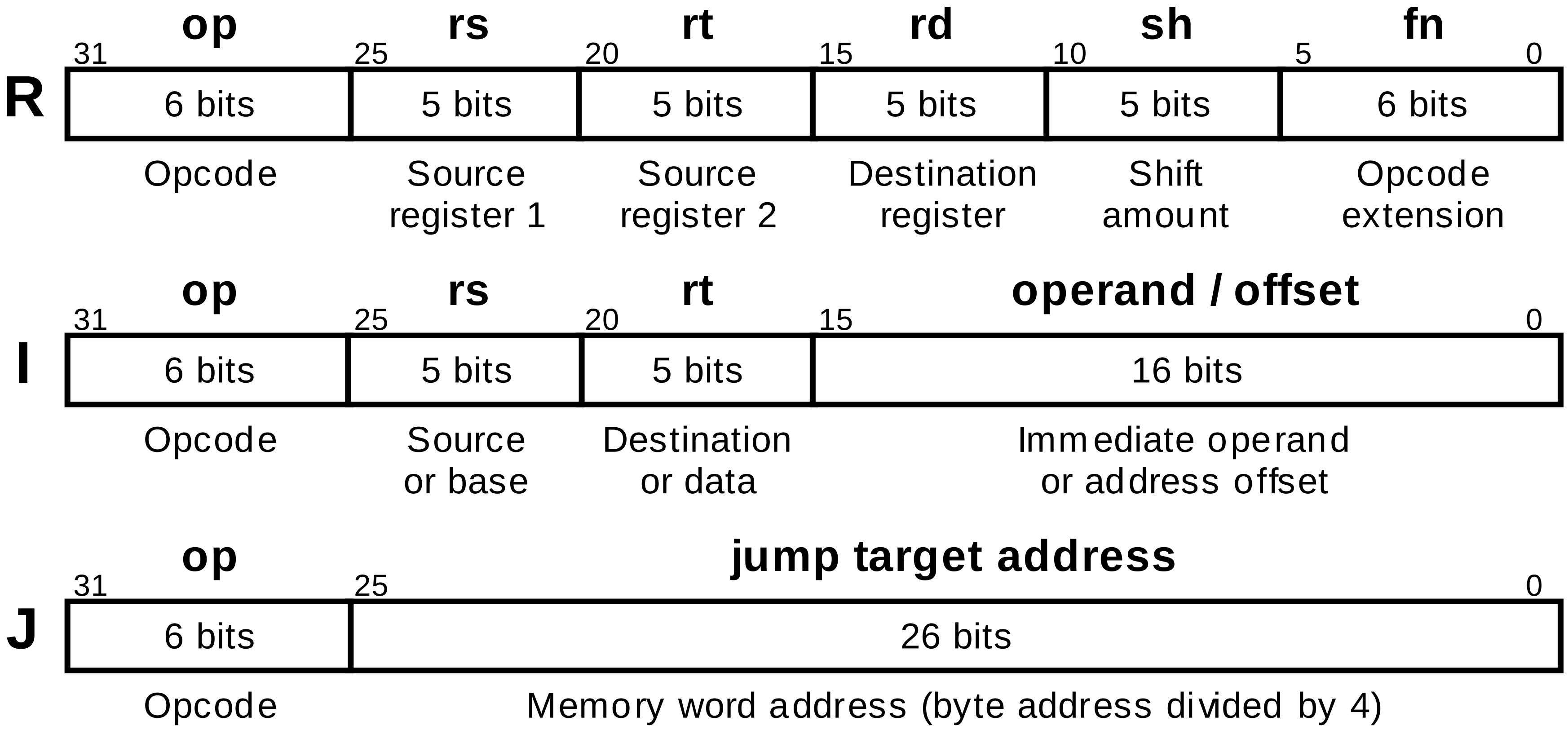
Decimal and hex constants

Decimal	25, 123456, –2873
Hexadecimal	0x59, 0x12b4c6, 0xffff0000

Machine instruction typically contains

an opcode
one or more source operands
possibly a destination operand

MiniMIPS Instruction Formats



R
op rs rt rd sh fn
31 25 20 15 10 5 0
6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits
Opcode | Source register 1 | Source register 2 | Destination register | Shift amount | Opcode extension

I
op rs rt operand / offset
31 25 20 15 0
6 bits | 5 bits | 5 bits | 16 bits
Opcode | Source or base | Destination or data | Immediate operand or address offset

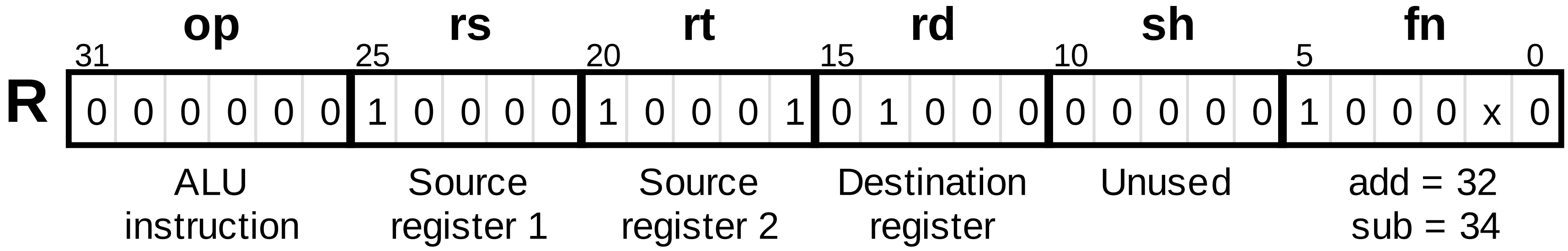
J
op jump target address
31 25 0
6 bits | 26 bits
Opcode | Memory word address (byte address divided by 4)

Figure 5.4 MiniMIPS instructions come in only three formats: register (R), immediate (I), and jump (J).

5.3 Simple Arithmetic/Logic Instructions

Add and subtract already discussed; logical instructions are similar

```
add  $t0,$s0,$s1     # set $t0 to ($s0)+($s1)
sub  $t0,$s0,$s1     # set $t0 to ($s0)-($s1)
and  $t0,$s0,$s1     # set $t0 to ($s0)∧($s1)
or   $t0,$s0,$s1     # set $t0 to ($s0)∨($s1)
xor  $t0,$s0,$s1     # set $t0 to ($s0)⊕($s1)
nor  $t0,$s0,$s1     # set $t0 to (($s0)∨($s1))′
```



	op	rs	rt	rd	sh	fn
	31	25	20	15	10	5 … 0
R	0 0 0 0 0 0	1 0 0 0 0	1 0 0 0 1	0 1 0 0 0	0 0 0 0 0	1 0 0 0 x 0
	ALU instruction	Source register 1	Source register 2	Destination register	Unused	add = 32 sub = 34

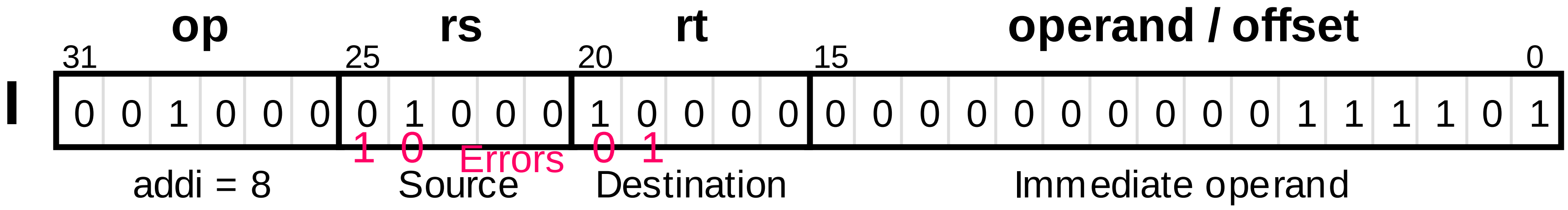
Figure 5.5 The arithmetic instructions add and sub have a format that is common to all two-operand ALU instructions. For these, the fn field specifies the arithmetic/logic operation to be performed.

Arithmetic/Logic with One Immediate Operand

An operand in the range [–32 768, 32 767], or [`0x0000`, `0xffff`], can be specified in the immediate field.

```
addi  $t0,$s0,61     # set $t0 to ($s0)+61
andi  $t0,$s0,61     # set $t0 to ($s0)∧61
ori   $t0,$s0,61     # set $t0 to ($s0)∨61
xori  $t0,$s0,0x00ff # set $t0 to ($s0)⊕ 0x00ff
```

For arithmetic instructions, the immediate operand is sign-extended



	op	rs	rt	operand / offset
	31	25	20	15 … 0
I	0 0 1 0 0 0	0 1 0 0 0	1 0 0 0 0	0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 1
	addi = 8	Source	Destination	Immediate operand

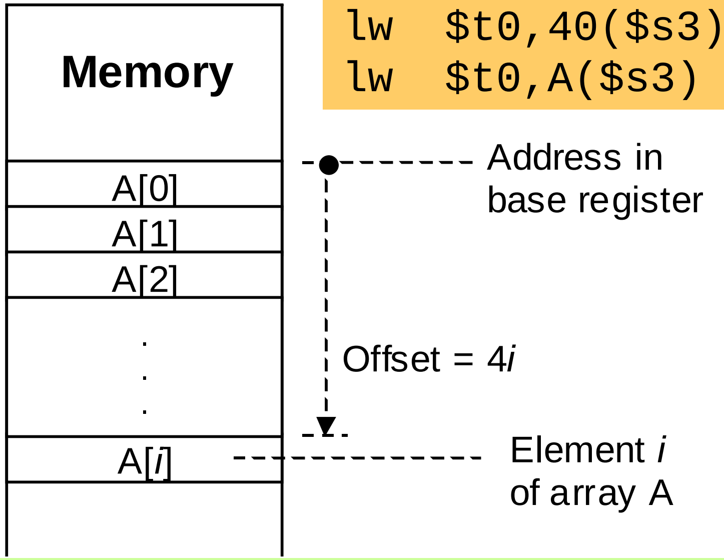
Figure 5.6 Instructions such as `addi` allow us to perform an arithmetic or logic operation for which one operand is a small constant.

5.4 Load and Store Instructions

	op	rs	rt	operand / offset
	31	25	20	15 0
I	1 0 x 0 1 1	1 0 0 1 1	0 1 0 0 0	0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0

	lw = 35
	sw = 43
		Base register	Data register	Offset relative to base

```
lw  $t0,40($s3)
lw  $t0,A($s3)
```



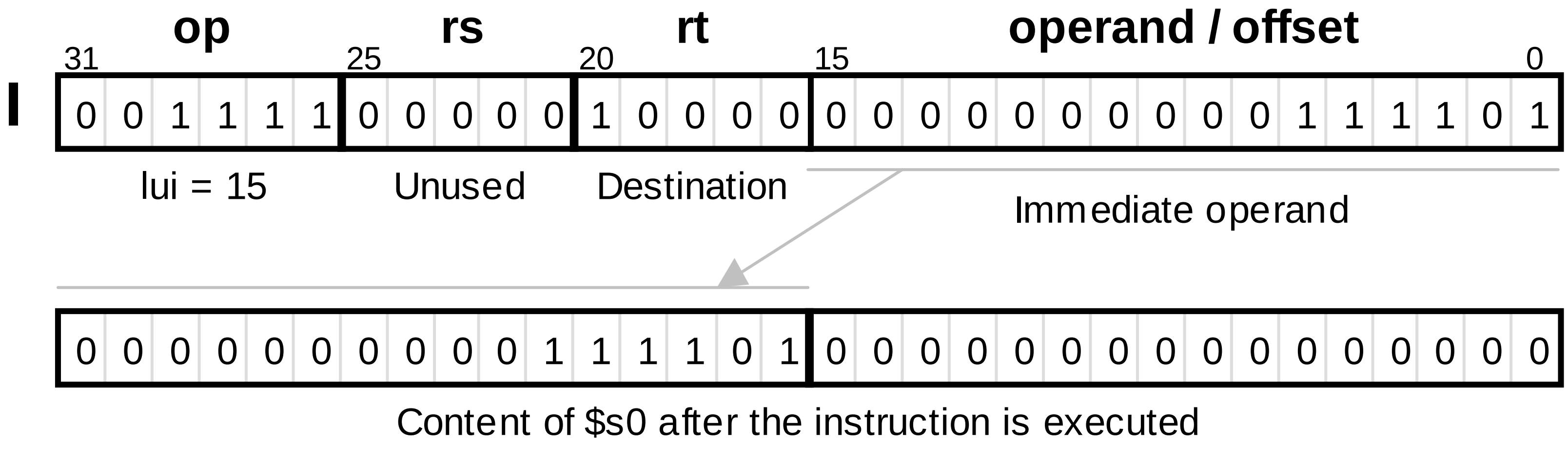
Note on base and offset:

The memory address is the sum of (`rs`) and an immediate value. Calling one of these the base and the other the offset is quite arbitrary. It would make perfect sense to interpret the address `A($s3)` as having the base A and the offset `($s3)`. However, a 16-bit base confines us to a small portion of memory space.

Figure 5.7 MiniMIPS `lw` and `sw` instructions and their memory addressing convention that allows for simple access to array elements via a base address and an offset (offset = 4*i* leads us to the *i*th word).

lw, **sw**, and **lui** Instructions

```
lw   $t0,40($s3)      # load mem[40+($s3)] in $t0
sw   $t0,A($s3)       # store ($t0) in mem[A+($s3)]
                      # “($s3)” means “content of $s3”
lui  $s0,61           # The immediate value 61 is
                      # loaded in upper half of $s0
                      # with lower 16b set to 0s
```



	op	rs	rt	operand / offset
	31	25	20	15 … 0
I	0 0 1 1 1 1	0 0 0 0 0	1 0 0 0 0	0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 1
	lui = 15	Unused	Destination	Immediate operand

0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Content of $s0 after the instruction is executed

Figure 5.8 The `lui` instruction allows us to load an arbitrary 16-bit value into the upper half of a register while setting its lower half to 0s.

Initializing a Register

Example 5.2

Show how each of these bit patterns can be loaded into `$s0`:

```
0010 0001 0001 0000 0000 0000 0011 1101
1111 1111 1111 1111 1111 1111 1111 1111
```

Solution

The first bit pattern has the hex representation: `0x2110003d`

```
lui  $s0,0x2110      # put the upper half in $s0
ori  $s0,0x003d      # put the lower half in $s0
```

Same can be done, with immediate values changed to `0xffff` for the second bit pattern. But, the following is simpler and faster:

```
nor  $s0,$zero,$zero # because (0 ∨ 0)′ = 1
```

5.5 Jump and Branch Instructions

Unconditional jump and jump through register instructions

```
j    verify         # go to mem loc named “verify”
jr   $ra            # go to address that is in $ra;
                    # $ra may hold a return address
```

$ra is the symbolic name for reg. $31 (return address)

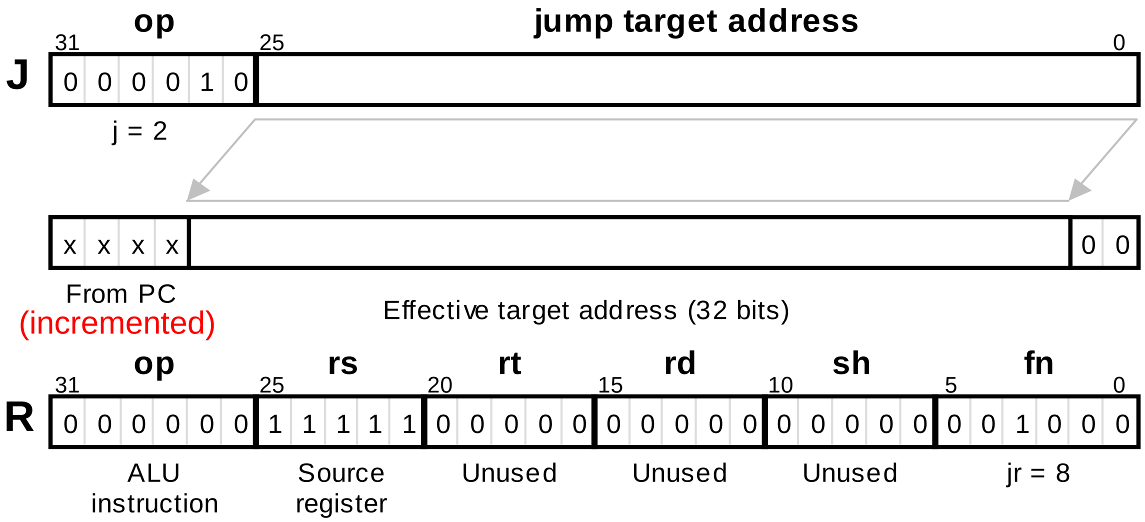


Figure 5.9 The jump instruction `j` of MiniMIPS is a J-type instruction which is shown along with how its effective target address is obtained. The jump register (`jr`) instruction is R-type, with its specified register often being `$ra`.

Conditional Branch Instructions

Conditional branches use PC-relative addressing

```
bltz $s1,L             # branch on ($s1)< 0
beq  $s1,$s2,L         # branch on ($s1)=($s2)
bne  $s1,$s2,L         # branch on ($s1)≠($s2)
```

	op	rs	rt	operand / offset
	31	25	20	15 0
I	0 0 0 0 0 1	1 0 0 0 1	0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 1
	bltz = 1	Source	Zero	Relative branch distance in words

	op	rs	rt	operand / offset
	31	25	20	15 0
I	0 0 0 1 0 x	1 0 0 0 1	1 0 0 1 0	0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 1

	beq = 4
	bne = 5
		Source 1	Source 2	Relative branch distance in words

Figure 5.10 (part 1) Conditional branch instructions of MiniMIPS.

Comparison Instructions for Conditional Branching

```
slt   $s1,$s2,$s3     # if ($s2)<($s3), set $s1 to 1
                      # else set $s1 to 0;
                      # often followed by beq/bne
slti  $s1,$s2,61      # if ($s2)<61, set $s1 to 1
                      # else set $s1 to 0
```

	op	rs	rt	rd	sh	fn
	31	25	20	15	10	5 0
R	0 0 0 0 0 0	1 0 0 1 0	1 0 0 1 1	1 0 0 0 1	0 0 0 0 0	1 0 1 0 1 0
	ALU instruction	Source 1 register	Source 2 register	Destination	Unused	slt = 42

	op	rs	rt	operand / offset
	31	25	20	15 0
I	0 0 1 0 1 0	1 0 0 1 0	1 0 0 0 1	0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 1
	slti = 10	Source	Destination	Immediate operand

Figure 5.10 (part 2) Comparison instructions of MiniMIPS.

Examples for Conditional Branching

If the branch target is too far to be reachable with a 16-bit offset (rare occurrence), the assembler automatically replaces the branch instruction `beq  $s0,$s1,L1` with:

```
      bne  $s1,$s2,L2      # skip jump if (s1)≠(s2)
      j    L1              # goto L1 if (s1)=(s2)
 L2: ...
```

Forming if-then constructs; e.g., `if (i == j) x = x + y`

```
      bne  $s1,$s2,endif  # branch on i≠j
      add  $t1,$t1,$t2    # execute the “then” part
endif: ...
```

If the condition were `(i < j)`, we would change the first line to:

```
      slt  $t0,$s1,$s2    # set $t0 to 1 if i<j
      beq  $t0,$0,endif   # branch if ($t0)=0;
                          # i.e., i not< j or i≥j
```

Compiling if-then-else Statements

Example 5.3

Show a sequence of MiniMIPS instructions corresponding to:

```
if (i<=j) x = x+1; z = 1; else y = y–1; z = 2*z
```

Solution

Similar to the “if-then” statement, but we need instructions for the “else” part and a way of skipping the “else” part after the “then” part.

```
      slt  $t0,$s2,$s1     # j<i? (inverse condition)
      bne  $t0,$zero,else  # if j<i goto else part
      addi $t1,$t1,1       # begin then part: x = x+1
      addi $t3,$zero,1     # z = 1
      j    endif           # skip the else part
else: addi $t2,$t2,-1      # begin else part: y = y–1
      add  $t3,$t3,$t3     # z = z+z
endif:...
```

5.6 Addressing Modes

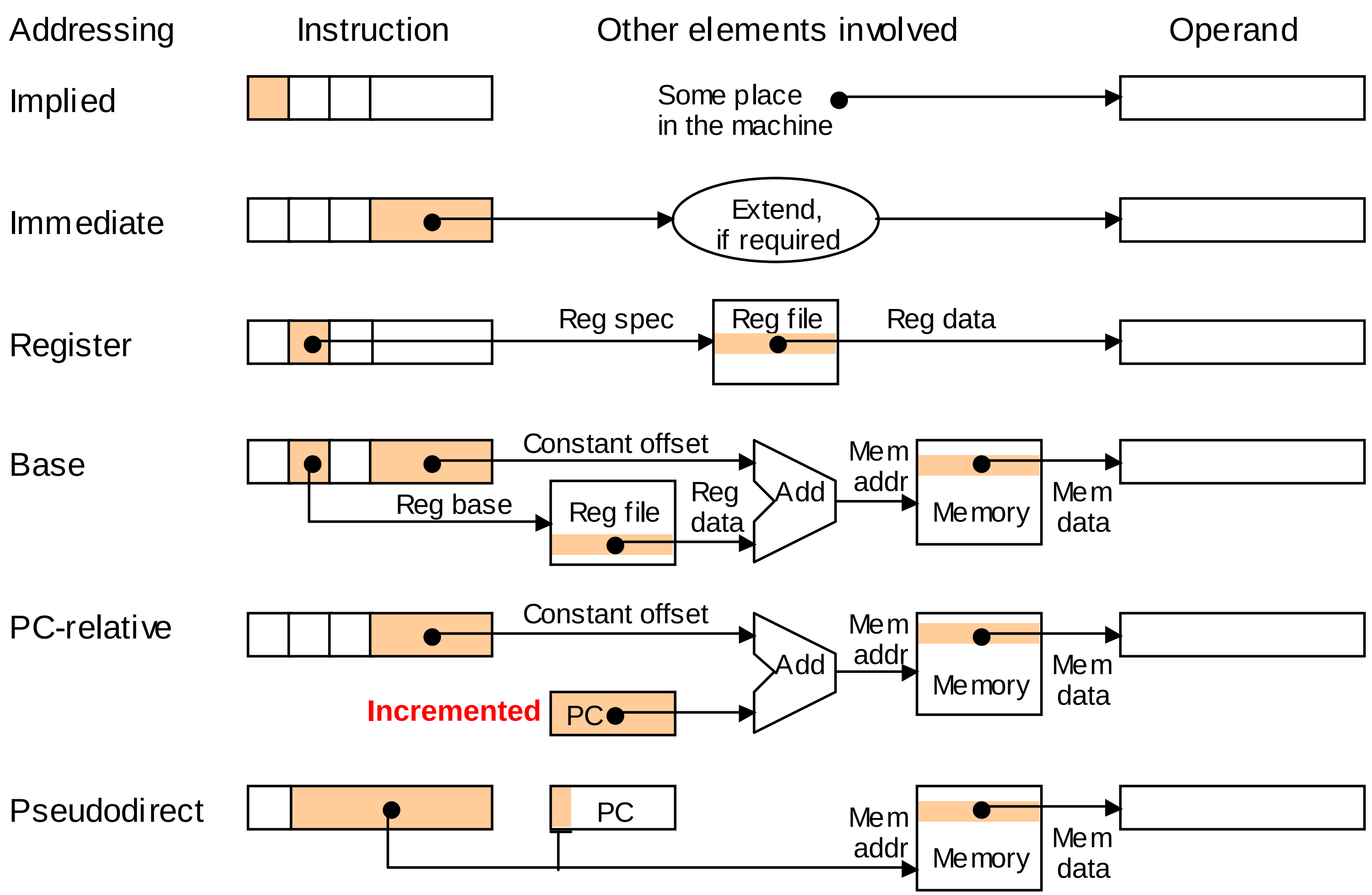


Figure 5.11 Schematic representation of addressing modes in MiniMIPS.

Finding the Maximum Value in a List of Integers

Example 5.5

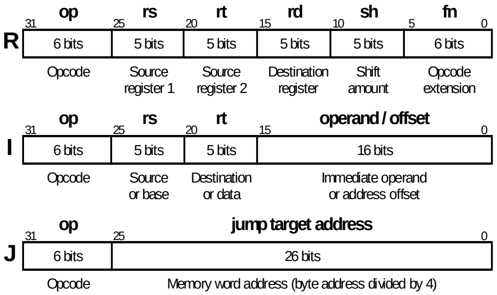
List A is stored in memory beginning at the address given in `$s1`.
List length is given in `$s2`.
Find the largest integer in the list and copy it into `$t0`.

Solution

Scan the list, holding the largest element identified thus far in `$t0`.

```
       lw    $t0,0($s1)       # initialize maximum to A[0]
       addi  $t1,$zero,0      # initialize index i to 0
loop:  add   $t1,$t1,1        # increment index i by 1
       beq   $t1,$s2,done     # if all elements examined, quit
       add   $t2,$t1,$t1      # compute 2i in $t2
       add   $t2,$t2,$t2      # compute 4i in $t2
       add   $t2,$t2,$s1      # form address of A[i] in $t2
       lw    $t3,0($t2)       # load value of A[i] into $t3
       slt   $t4,$t0,$t3      # maximum < A[i]?
       beq   $t4,$zero,loop   # if not, repeat with no change
       addi  $t0,$t3,0               #  if  so,  A[i]  is  the  new
   maximum
       j     loop             # change completed; now repeat
done:  ...                    # continuation of the program
```

The 20 MiniMIPS Instructions Covered So Far



R: op (31) 6 bits Opcode | rs (25) 5 bits Source register 1 | rt (20) 5 bits Source register 2 | rd (15) 5 bits Destination register | sh (10) 5 bits Shift amount | fn (5) 6 bits Opcode extension (0)

I: op (31) 6 bits Opcode | rs (25) 5 bits Source or base | rt (20) 5 bits Destination or data | operand / offset (15) 16 bits Immediate operand or address offset (0)

J: op (31) 6 bits Opcode | jump target address (25) 26 bits Memory word address (byte address divided by 4) (0)

	Instruction	Usage	op	fn
Copy	Load upper immediate	`lui rt,imm`	15	
Arithmetic	Add	`add rd,rs,rt`	0	32
	Subtract	`sub rd,rs,rt`	0	34
	Set less than	`slt rd,rs,rt`	0	42
	Add immediate	`addi rt,rs,imm`	8	
	Set less than immediate	`slti rd,rs,imm`	10	
Logic	AND	`and rd,rs,rt`	0	36
	OR	`or rd,rs,rt`	0	37
	XOR	`xor rd,rs,rt`	0	38
	NOR	`nor rd,rs,rt`	0	39
	AND immediate	`andi rt,rs,imm`	12	
	OR immediate	`ori rt,rs,imm`	13	
	XOR immediate	`xori rt,rs,imm`	14	
Memory access	Load word	`lw rt,imm(rs)`	35	
	Store word	`sw rt,imm(rs)`	43	
Control transfer	Jump	`j L`	2	
	Jump register	`jr rs`	0	8
	Branch less than 0	`bltz rs,L`	1	
	Branch equal	`beq rs,rt,L`	4	
	Branch not equal	`bne rs,rt,L`	5	

Table 5.1

6 Procedures and Data

Finish our study of MiniMIPS instructions and its data types:

- Instructions for procedure call/return, misc. instructions
- Procedure parameters and results, utility of stack

Topics in This Chapter
6.1 Simple Procedure Calls
6.2 Using the Stack for Data Storage
6.3 Parameters and Results
6.4 Data Types
6.5 Arrays and Pointers
6.6 Additional Instructions

6.1 Simple Procedure Calls

Using a procedure involves the following sequence of actions:

1. Put arguments in places known to procedure (reg’s `$a0-$a3`)
2. Transfer control to procedure, saving the return address (`jal`)
3. Acquire storage space, if required, for use by the procedure
4. Perform the desired task
5. Put results in places known to calling program (reg’s `$v0-$v1`)
6. Return control to calling point (`jr`)

MiniMIPS instructions for procedure call and return from procedure:

```
jal  proc   # jump to loc “proc” and link;
            # “link” means “save the return
            # address” (PC)+4 in $ra ($31)

jr   rs     # go to loc addressed by rs
```

Illustrating a Procedure Call

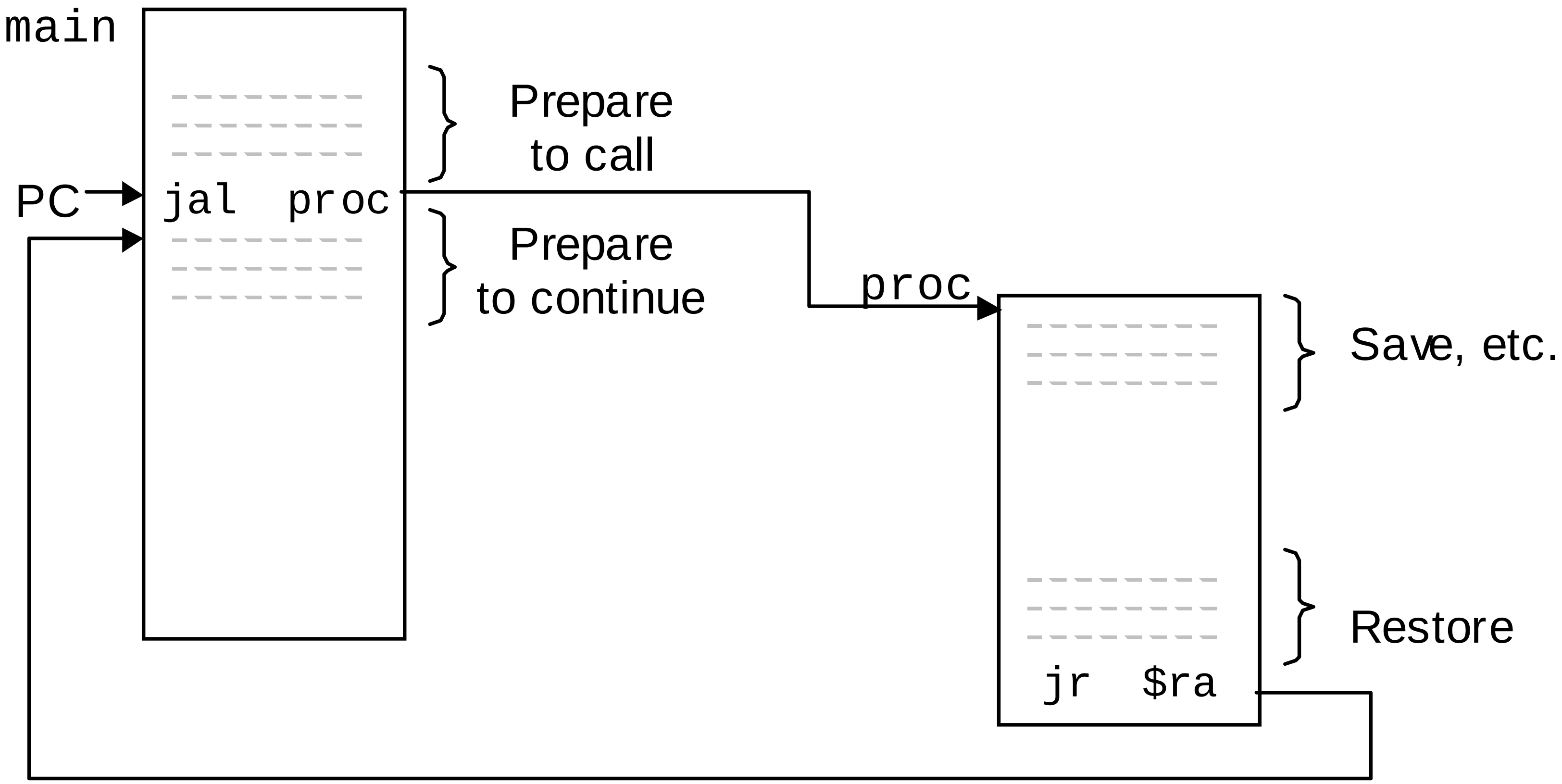
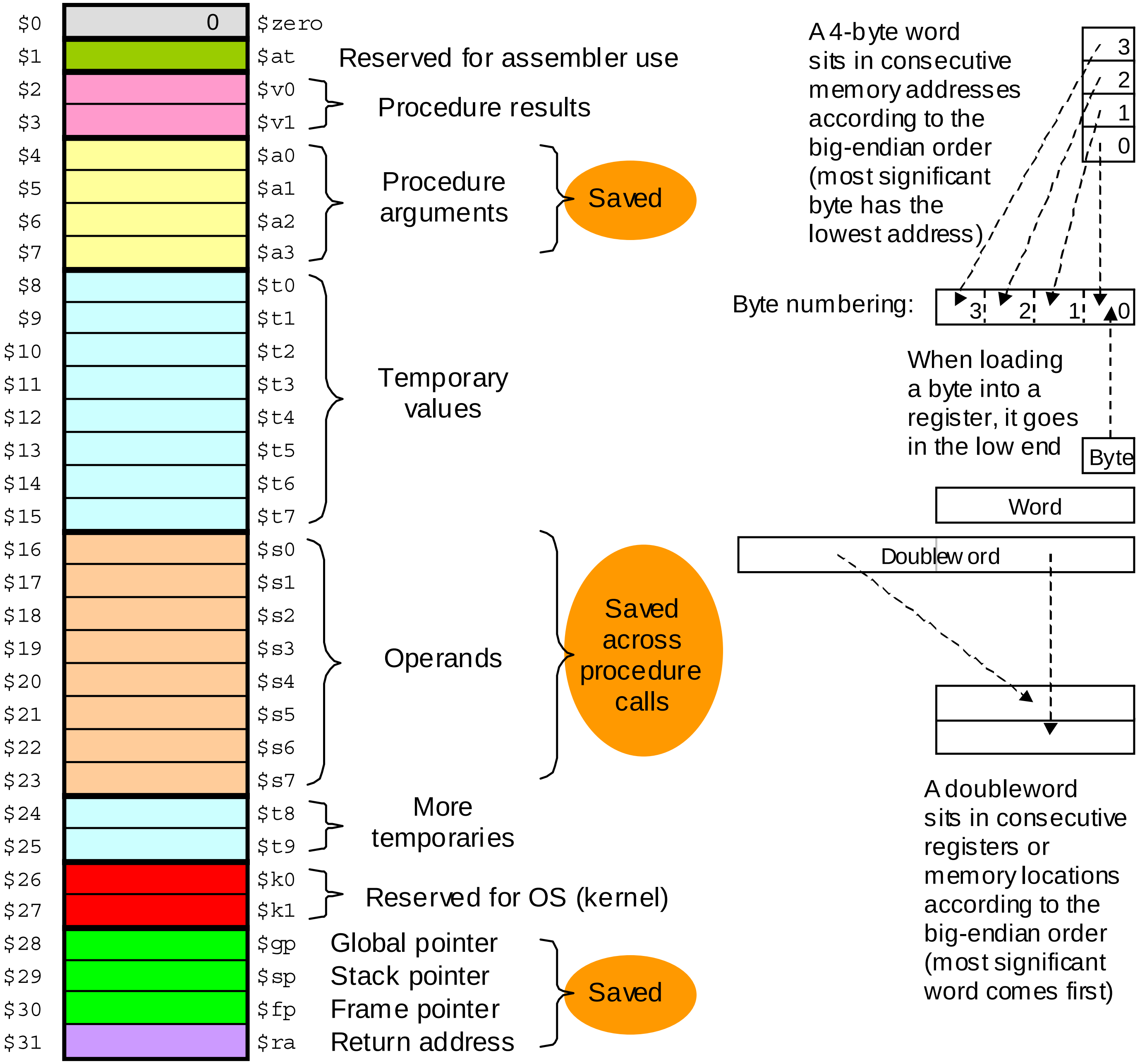


Figure 6.1 Relationship between the main program and a procedure.

Recalling Register Conventions



A 4-byte word sits in consecutive memory addresses according to the big-endian order (most significant byte has the lowest address)

When loading a byte into a register, it goes in the low end

A doubleword sits in consecutive registers or memory locations according to the big-endian order (most significant word comes first)

Figure 5.2 Registers and data sizes in MiniMIPS.

A Simple MiniMIPS Procedure

Example 6.1

Procedure to find the absolute value of an integer.

```
$v0 ← |($a0)|
```

Solution

The absolute value of *x* is –*x* if *x* < 0 and *x* otherwise.

```
 abs: sub  $v0,$zero,$a0  # put -($a0) in $v0;
                          # in case ($a0) < 0
      bltz $a0,done       # if ($a0)<0 then done
      add  $v0,$a0,$zero  # else put ($a0) in $v0
done: jr   $ra            # return to calling program
```

In practice, we seldom use such short procedures because of the overhead that they entail. In this example, we have 3-4 instructions of overhead for 3 instructions of useful computation.

Nested Procedure Calls

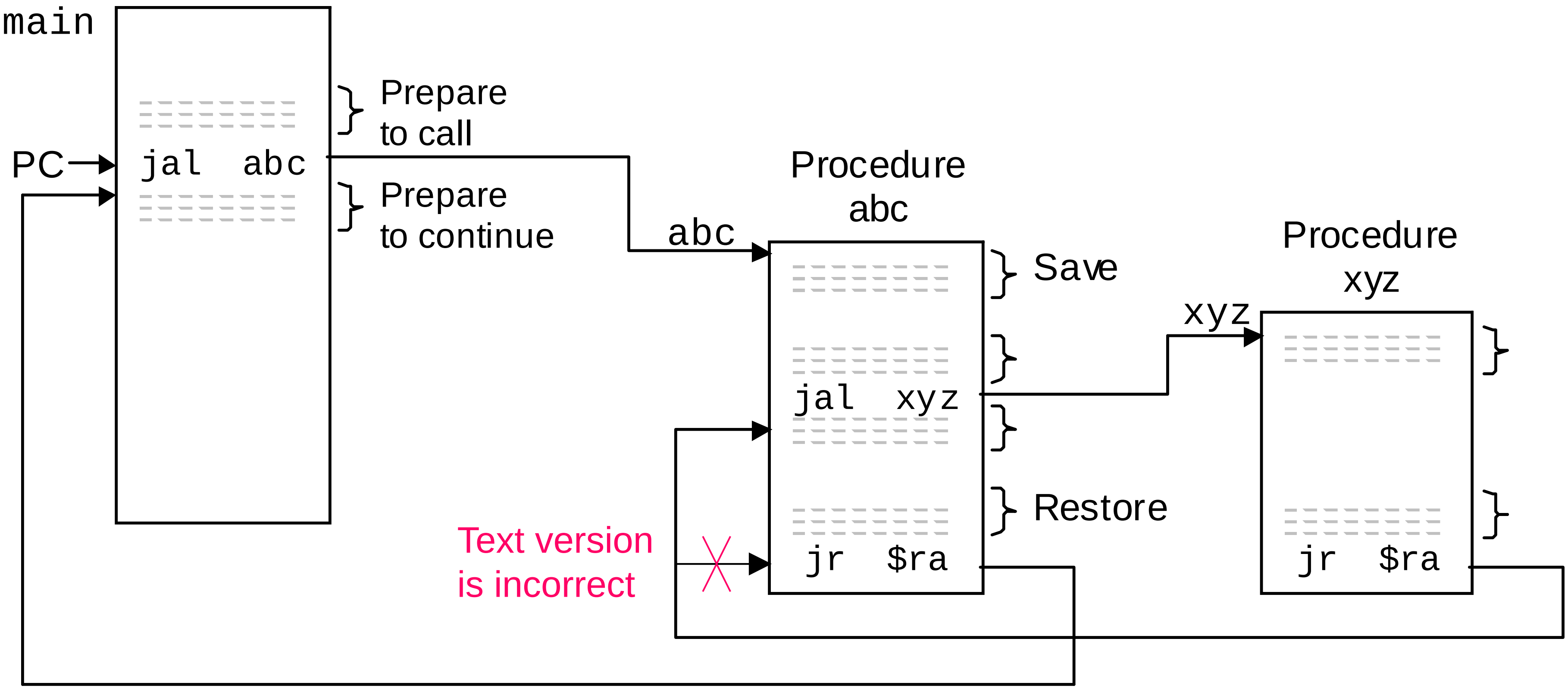
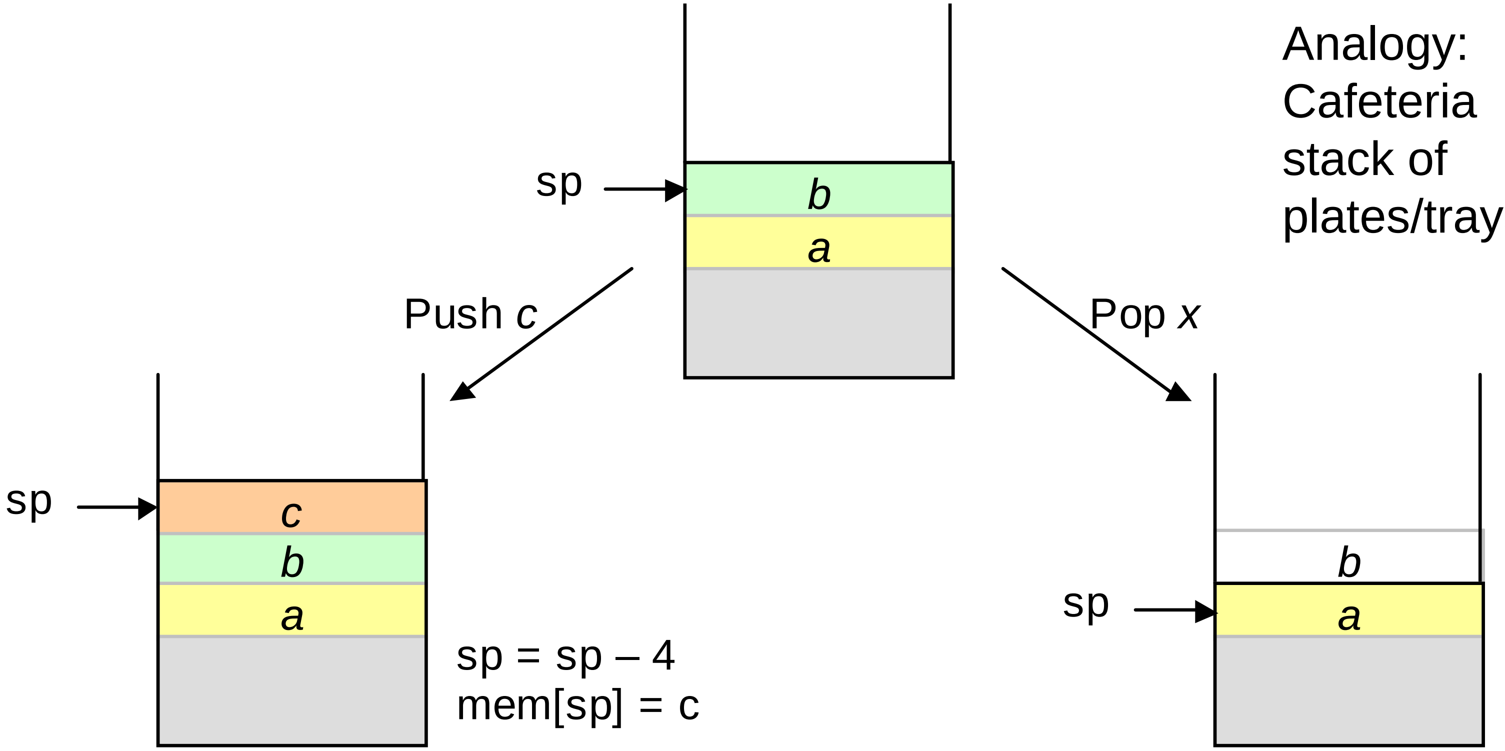


Figure 6.2 Example of nested procedure calls.

6.2 Using the Stack for Data Storage



Analogy: Cafeteria stack of plates/trays

Figure 6.4 Effects of push and pop operations on a stack.

```
push: addi  $sp,$sp,-4
      sw    $t4,0($sp)
```

```
pop: lw    $t5,0($sp)
     addi  $sp,$sp,4
```

Memory Map in MiniMIPS

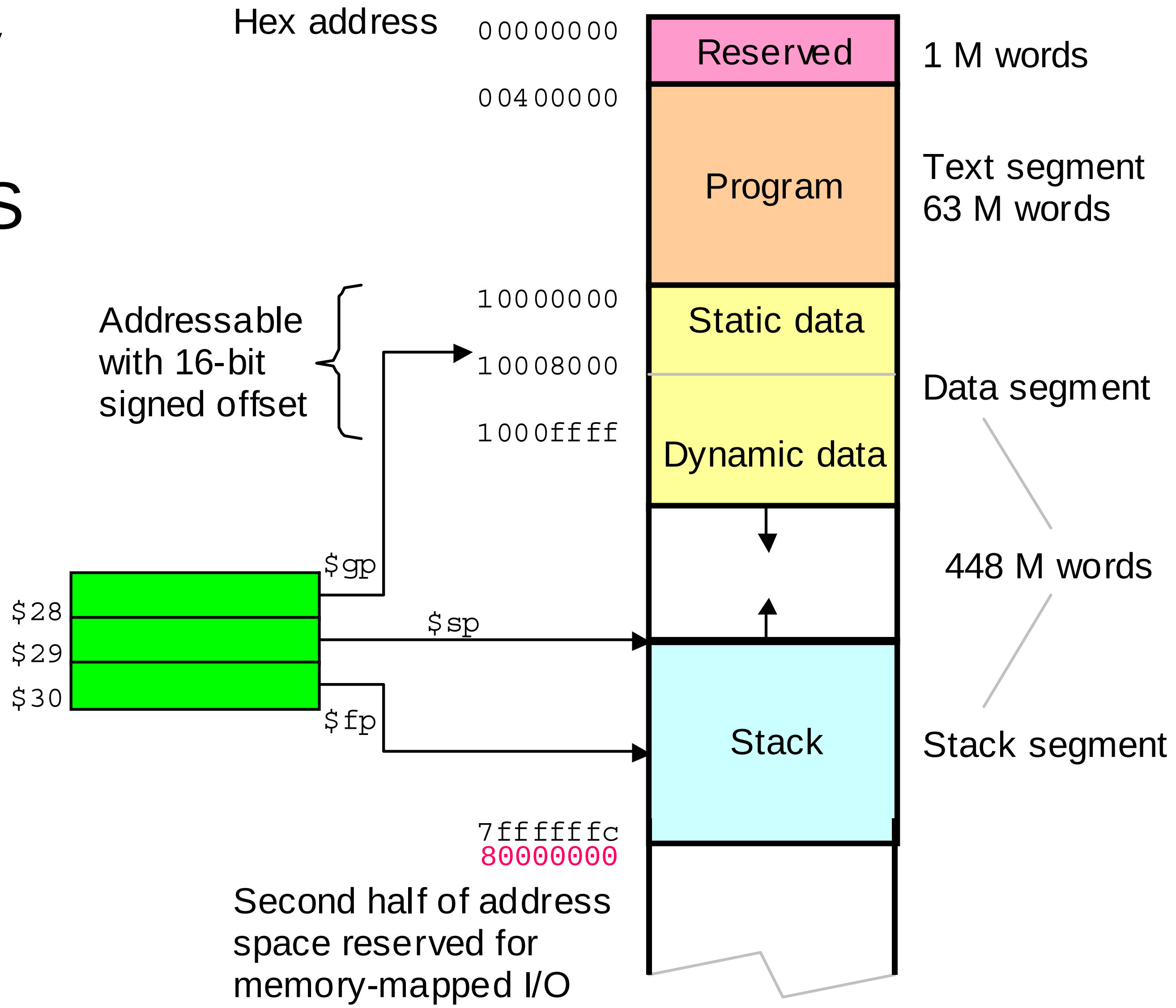


Figure 6.3 Overview of the memory address space in MiniMIPS.

6.3 Parameters and Results

Stack allows us to pass/return an arbitrary number of values

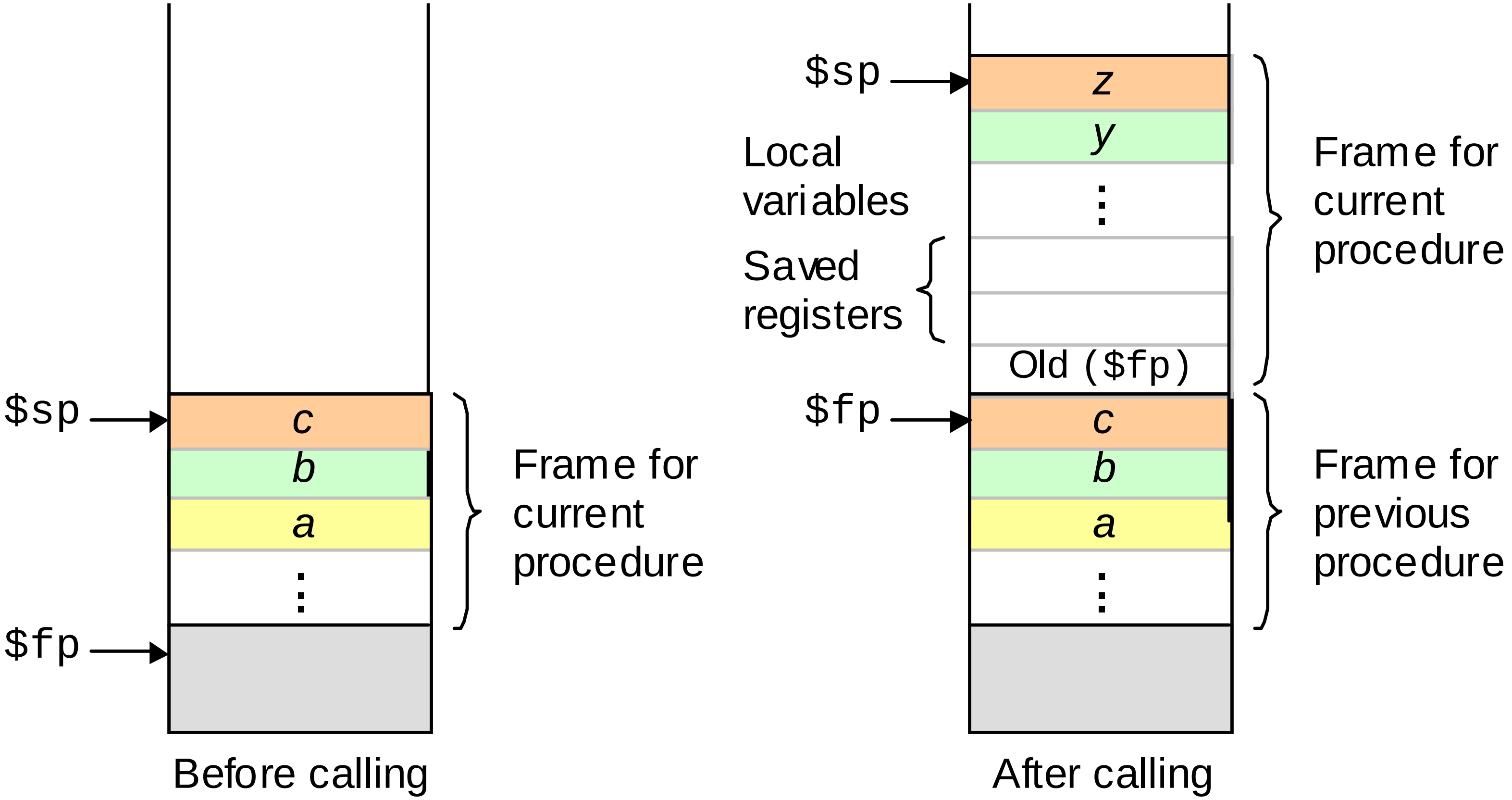
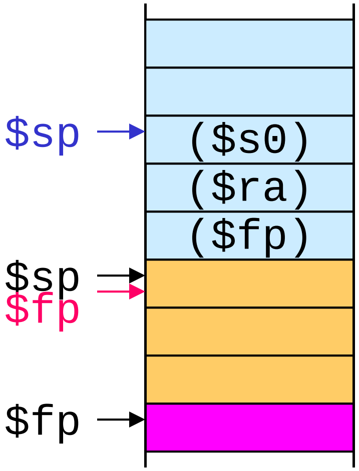


Figure 6.5 Use of the stack by a procedure.

Example of Using the Stack

Saving `$fp`, `$ra`, and `$s0` onto the stack and restoring them at the end of the procedure



```
proc: sw   $fp,-4($sp)    # save the old frame pointer
      addi $fp,$sp,0      # save ($sp) into $fp
      addi $sp,$sp,–12    # create 3 spaces on top of stack
      sw   $ra,-8($fp)    # save ($ra) in 2nd stack element
      sw   $s0,-12($fp)   # save ($s0) in top stack element
        .
        .
        .
      lw   $s0,-12($fp)   # put top stack element in $s0
      lw   $ra,-8($fp)    # put 2nd stack element in $ra
      addi $sp,$fp, 0     # restore $sp to original state
      lw   $fp,-4($sp)    # restore $fp to original state
      jr   $ra            # return from procedure
```

6.4 Data Types

Data size (number of bits), data type (meaning assigned to bits)

Signed integer:	byte	word	
Unsigned integer:	byte	word	
Floating-point number:		word	doubleword
Bit string:	byte	word	doubleword

Converting from one size to another

Type	**8-bit number**	**Value**	**32-bit version of the number**
Unsigned	0010 1011	43	0000 0000 0000 0000 0000 0000 0010 1011
Unsigned	1010 1011	171	0000 0000 0000 0000 0000 0000 1010 1011
Signed	0010 1011	+43	0000 0000 0000 0000 0000 0000 0010 1011
Signed	1010 1011	–85	1111 1111 1111 1111 1111 1111 1010 1011

ASCII Characters

Table 6.1 ASCII (American standard code for information interchange)

	0	1	2	3	4	5	6	7	8-9	a-f
0	NUL	DLE	SP	0	@	P	`	p	More controls	More symbols
1	SOH	DC1	!	1	A	Q	a	q		
2	STX	DC2	“	2	B	R	b	r		
3	ETX	DC3	#	3	C	S	c	s		
4	EOT	DC4	$	4	D	T	d	t		
5	ENQ	NAK	%	5	E	U	e	u		
6	ACK	SYN	&	6	F	V	f	v		
7	BEL	ETB	‘	7	G	W	g	w		
8	BS	CAN	(	8	H	X	h	x		
9	HT	EM	)	9	I	Y	i	y		
a	LF	SUB	*	:	J	Z	j	z		
b	VT	ESC	+	;	K	[	k	{		
c	FF	FS	,	<	L	\	l	|		
d	CR	GS	-	=	M	]	m	}		
e	SO	RS	.	>	N	^	n	~		
f	SI	US	/	?	O	_	o	DEL		

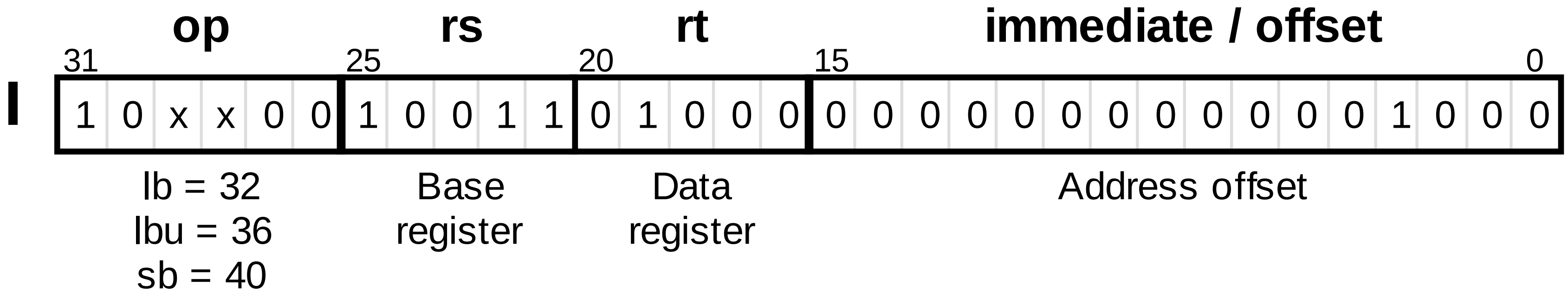
8-bit ASCII code (col #, row #)$_{\text{hex}}$

e.g., code for + is (2b) $_{\text{hex}}$ or (0010 1011)$_{\text{two}}$

Loading and Storing Bytes

Bytes can be used to store ASCII characters or small integers. MiniMIPS addresses refer to bytes, but registers hold words.

```
lb   $t0,8($s3)      # load rt with mem[8+($s3)]
                     # sign-extend to fill reg
lbu  $t0,8($s3)      # load rt with mem[8+($s3)]
                     # zero-extend to fill reg
sb   $t0,A($s3)      # LSB of rt to mem[A+($s3)]
```



	op	rs	rt	immediate / offset
	31	25	20	15 … 0
I	1 0 x x 0 0	1 0 0 1 1	0 1 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0

	lb = 32
	lbu = 36
	sb = 40
		Base register	Data register	Address offset

Figure 6.6 Load and store instructions for byte-size data elements.

Meaning of a Word in Memory

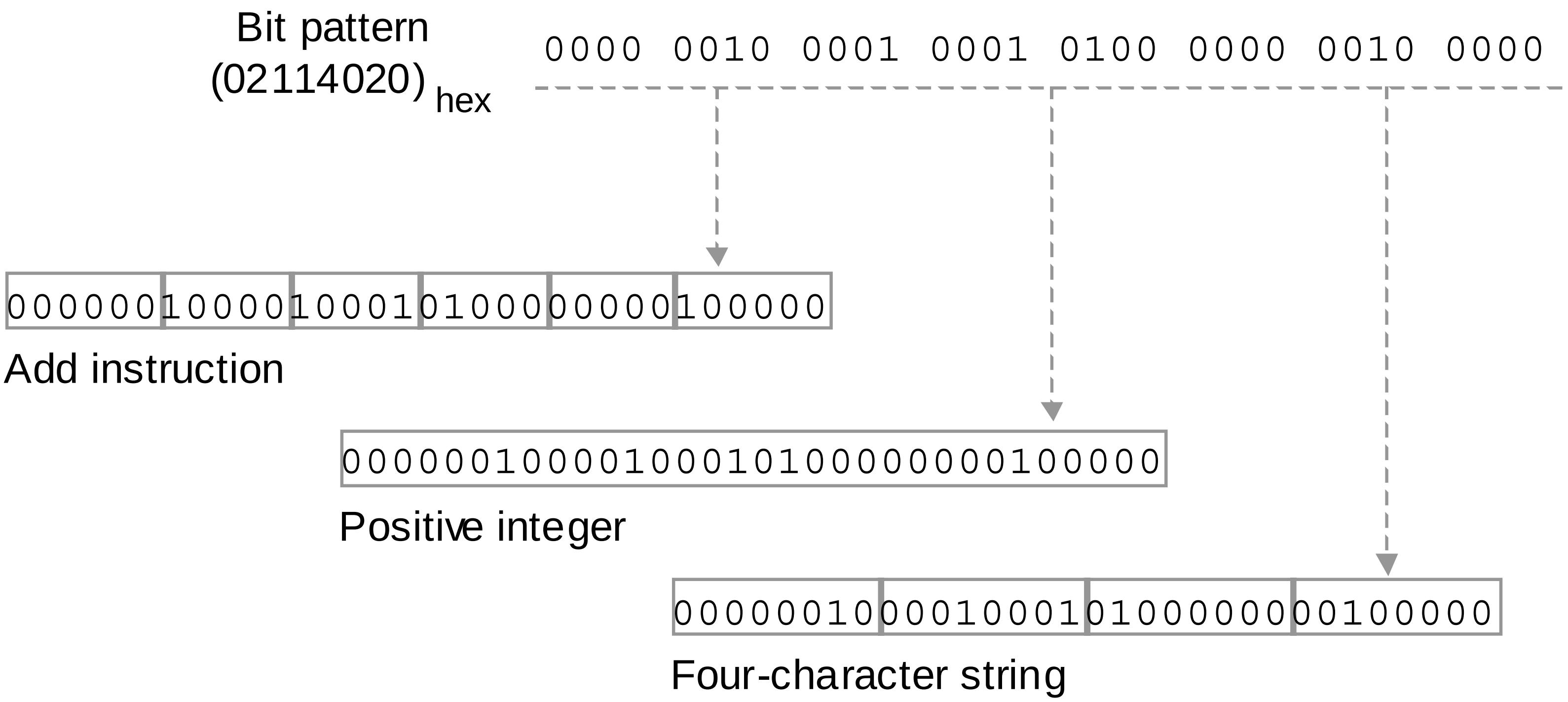


Figure 6.7 A 32-bit word has no inherent meaning and can be interpreted in a number of equally valid ways in the absence of other cues (e.g., context) for the intended meaning.

6.5 Arrays and Pointers

Index: Use a register that holds the index *i* and increment the register in each step to effect moving from element *i* of the list to element *i* + 1

Pointer: Use a register that points to (holds the address of) the list element being examined and update it in each step to point to the next element

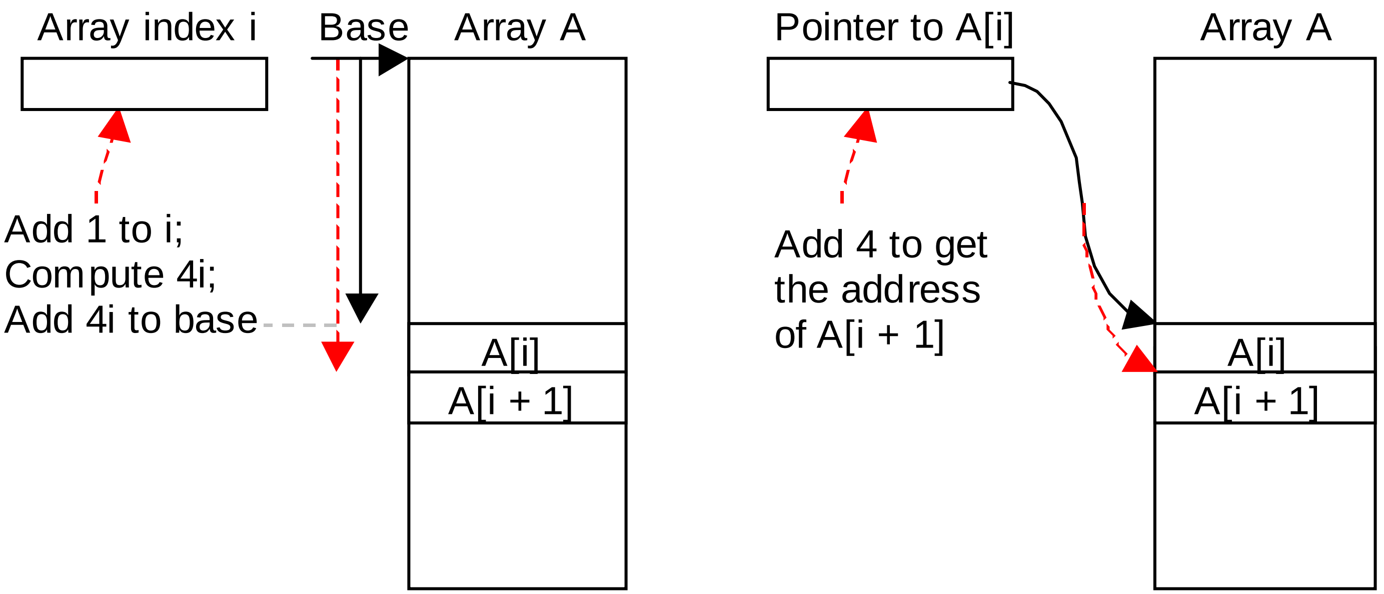


Figure 6.8 Stepping through the elements of an array using the indexing method and the pointer updating method.

Selection Sort

Example 6.4

To sort a list of numbers, repeatedly perform the following:
Find the max element, swap it with the last item, move up the “last” pointer

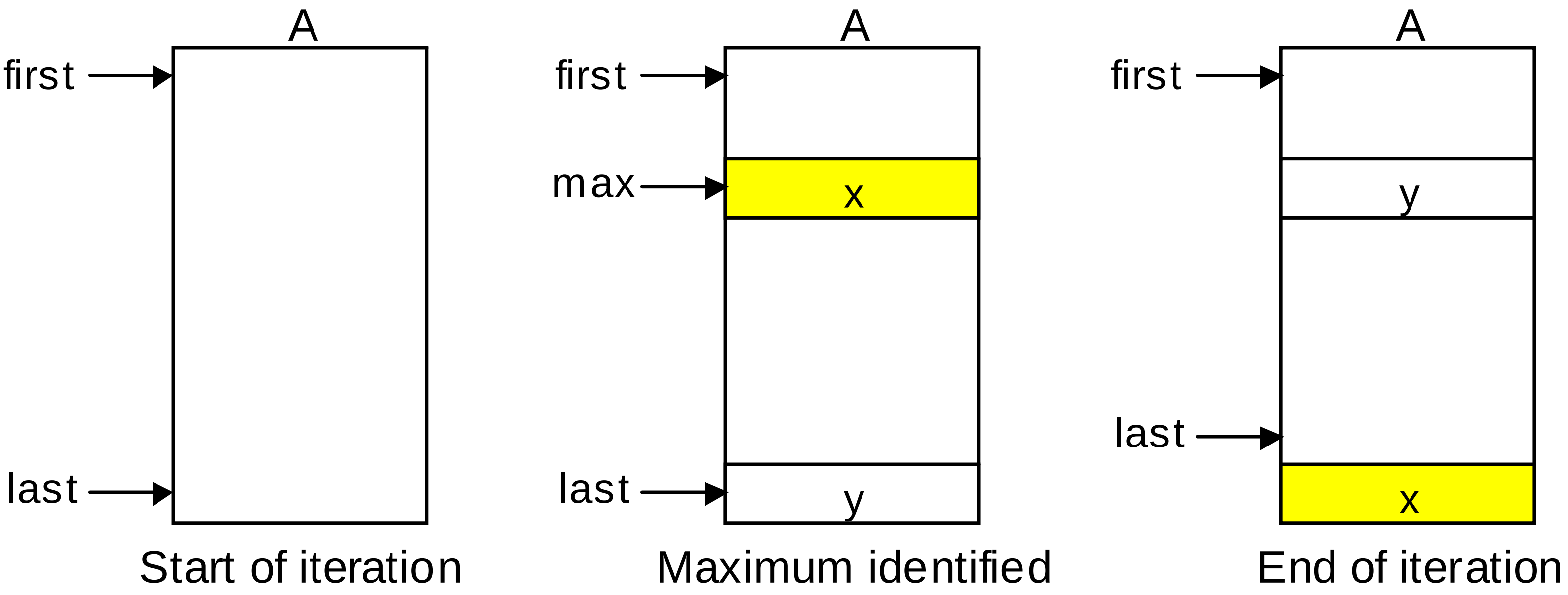
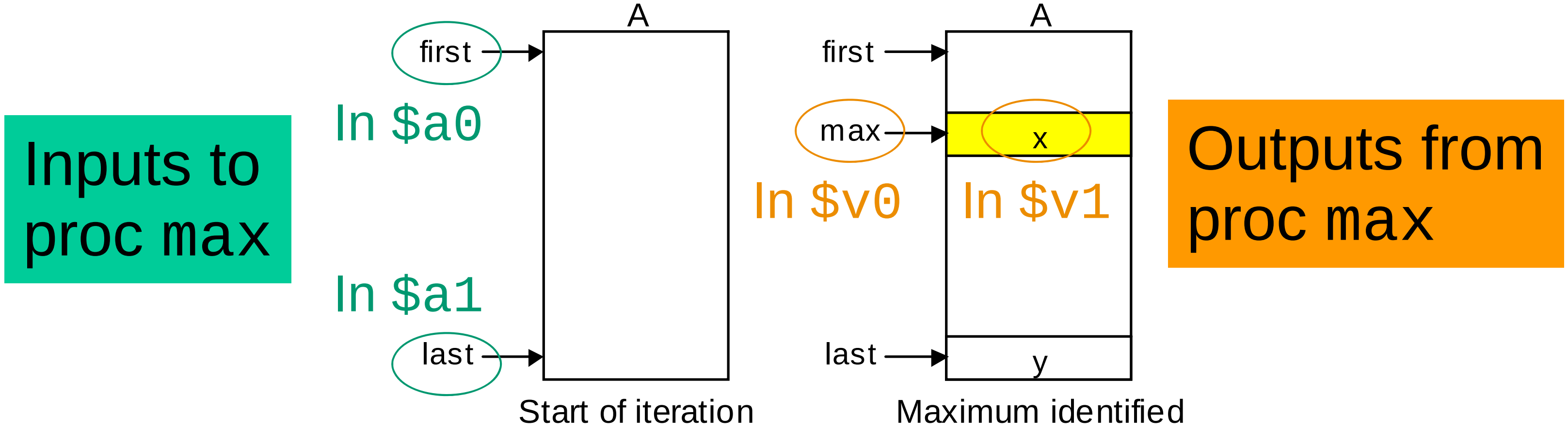


Figure 6.9 One iteration of selection sort.

Selection Sort Using the Procedure max

Example 6.4 (continued)



Inputs to proc max

In $a0

In $a1

In $v0

In $v1

Outputs from proc max

Start of iteration

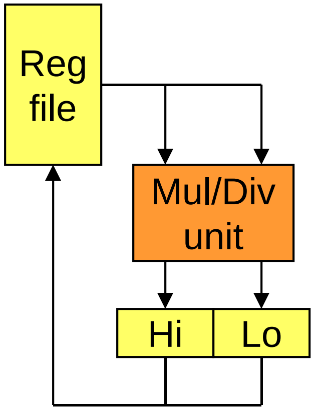
Maximum identified

```
sort: beq  $a0,$a1,done  # single-element list is sorted
      jal  max           # call the max procedure
      lw   $t0,0($a1)    # load last element into $t0
      sw   $t0,0($v0)    # copy the last element to max loc
      sw   $v1,0($a1)    # copy max value to last element
      addi $a1,$a1,-4    # decrement pointer to last element
      j    sort          # repeat sort for smaller list
done: ...                # continue with rest of program
```

6.6 Additional Instructions

MiniMIPS instructions for multiplication and division:

```
mult   $s0, $s1    # set Hi,Lo to ($s0)×($s1)
div    $s0, $s1    # set Hi to ($s0)mod($s1)
                   # and Lo to ($s0)/($s1)
mfhi   $t0         # set $t0 to (Hi)
mflo   $t0         # set $t0 to (Lo)
```



	op	rs	rt	rd	sh	fn
	31	25	20	15	10	5 0
R	0 0 0 0 0 0	1 0 0 0 0	1 0 0 0 1	0 0 0 0 0	0 0 0 0 0	0 1 1 0 x 0
	ALU instruction	Source register 1	Source register 2	Unused	Unused	mult = 24 div = 26

Figure 6.10 The multiply (`mult`) and divide (`div`) instructions of MiniMIPS.

	op	rs	rt	rd	sh	fn
	31	25	20	15	10	5 0
R	0 0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	0 1 0 0 0	0 0 0 0 0	0 1 0 0 x 0
	ALU instruction	Unused	Unused	Destination register	Unused	mfhi = 16 mflo = 18

Figure 6.11 MiniMIPS instructions for copying the contents of `Hi` and `Lo` registers into general registers .

Logical Shifts

MiniMIPS instructions for left and right shifting:

```
sll   $t0,$s1,2    # $t0=($s1) left-shifted by 2
srl   $t0,$s1,2    # $t0=($s1) right-shifted by 2
sllv  $t0,$s1,$s0  # $t0=($s1) left-shifted by ($s0)
srlv  $t0,$s1,$s0  # $t0=($s1) right-shifted by ($s0)
```

	op	rs	rt	rd	sh	fn
	31	25	20	15	10	5 0
R	0 0 0 0 0 0	0 0 0 0 0	1 0 0 0 1	0 1 0 0 0	0 0 0 1 0	0 0 0 0 x 0
	ALU instruction	Unused	Source register	Destination register	Shift amount	sll = 0 srl = 2

	op	rs	rt	rd	sh	fn
	31	25	20	15	10	5 0
R	0 0 0 0 0 0	1 0 0 0 0	1 0 0 0 1	0 1 0 0 0	0 0 0 0 0	0 0 0 1 x 0
	ALU instruction	Amount register	Source register	Destination register	Unused	sllv = 4 srlv = 6

Figure 6.12 The four logical shift instructions of MiniMIPS.

Unsigned Arithmetic and Miscellaneous Instructions

MiniMIPS instructions for unsigned arithmetic (no overflow exception):

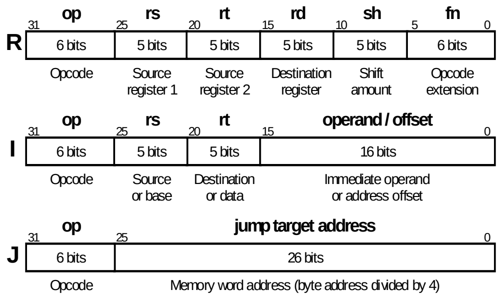
```
addu  $t0,$s0,$s1    # set $t0 to ($s0)+($s1)
subu  $t0,$s0,$s1    # set $t0 to ($s0)–($s1)
multu $s0,$s1        # set Hi,Lo to ($s0)×($s1)
divu  $s0,$s1        # set Hi to ($s0)mod($s1)
                     # and Lo to ($s0)/($s1)
addiu $t0,$s0,61     # set $t0 to ($s0)+61;
                     # the immediate operand is
                     # sign extended
```

To make MiniMIPS more powerful and complete, we introduce later:

```
sra   $t0,$s1,2      # sh. right arith (Sec. 10.5)
srav  $t0,$s1,$s0    # shift right arith variable
syscall              # system call (Sec. 7.6)
```

The 20 MiniMIPS Instructions from Chapter 6 (40 in all so far)

Table 6.2 (partial)



	Instruction	Usage	op	fn
Copy	Move from Hi	`mfhi rd`	0	16
	Move from Lo	`mflo rd`	0	18
Arithmetic	Add unsigned	`addu rd,rs,rt`	0	33
	Subtract unsigned	`subu rd,rs,rt`	0	35
	Multiply	`mult rs,rt`	0	24
	Multiply unsigned	`multu rs,rt`	0	25
	Divide	`div rs,rt`	0	26
	Divide unsigned	`divu rs,rt`	0	27
	Add immediate unsigned	`addiu rs,rt,imm`	9	
Shift	Shift left logical	`sll rd,rt,sh`	0	0
	Shift right logical	`srl rd,rt,sh`	0	2
	Shift right arithmetic	`sra rd,rt,sh`	0	3
	Shift left logical variable	`sllv rd,rt,rs`	0	4
	Shift right logical variable	`srlv rt,rd,rs`	0	6
	Shift right arith variable	`srav rd,rt,rd`	0	7
Memory access	Load byte	`lb rt,imm(rs)`	32	
	Load byte unsigned	`lbu rt,imm(rs)`	36	
	Store byte	`sb rt,imm(rs)`	40	
Control transfer	Jump and link	`jal L`	3	
	System call	`syscall`	0	12

Table 6.2 The 37 + 3 MiniMIPS Instructions Covered So Far

Instruction	Usage
Load upper immediate	lui rt,imm
Add	add rd,rs,rt
Subtract	sub rd,rs,rt
Set less than	slt rd,rs,rt
Add immediate	addi rt,rs,imm
Set less than immediate	slti rd,rs,imm
AND	and rd,rs,rt
OR	or rd,rs,rt
XOR	xor rd,rs,rt
NOR	nor rd,rs,rt
AND immediate	andi rt,rs,imm
OR immediate	ori rt,rs,imm
XOR immediate	xori rt,rs,imm
Load word	lw rt,imm(rs)
Store word	sw rt,imm(rs)
Jump	j L
Jump register	jr rs
Branch less than 0	bltz rs,L
Branch equal	beq rs,rt,L
Branch not equal	bne rs,rt,L
Move from Hi	mfhi rd
Move from Lo	mflo rd
Add unsigned	addu rd,rs,rt
Subtract unsigned	subu rd,rs,rt
Multiply	mult rs,rt
Multiply unsigned	multu rs,rt
Divide	div rs,rt
Divide unsigned	divu rs,rt
Add immediate unsigned	addiu rs,rt,imm
Shift left logical	sll rd,rt,sh
Shift right logical	srl rd,rt,sh
Shift right arithmetic	sra rd,rt,sh
Shift left logical variable	sllv rd,rt,rs
Shift right logical variable	srlv rd,rt,rs
Shift right arith variable	srav rd,rt,rs
Load byte	lb rt,imm(rs)
Load byte unsigned	lbu rt,imm(rs)
Store byte	sb rt,imm(rs)
Jump and link	jal L
System call	syscall

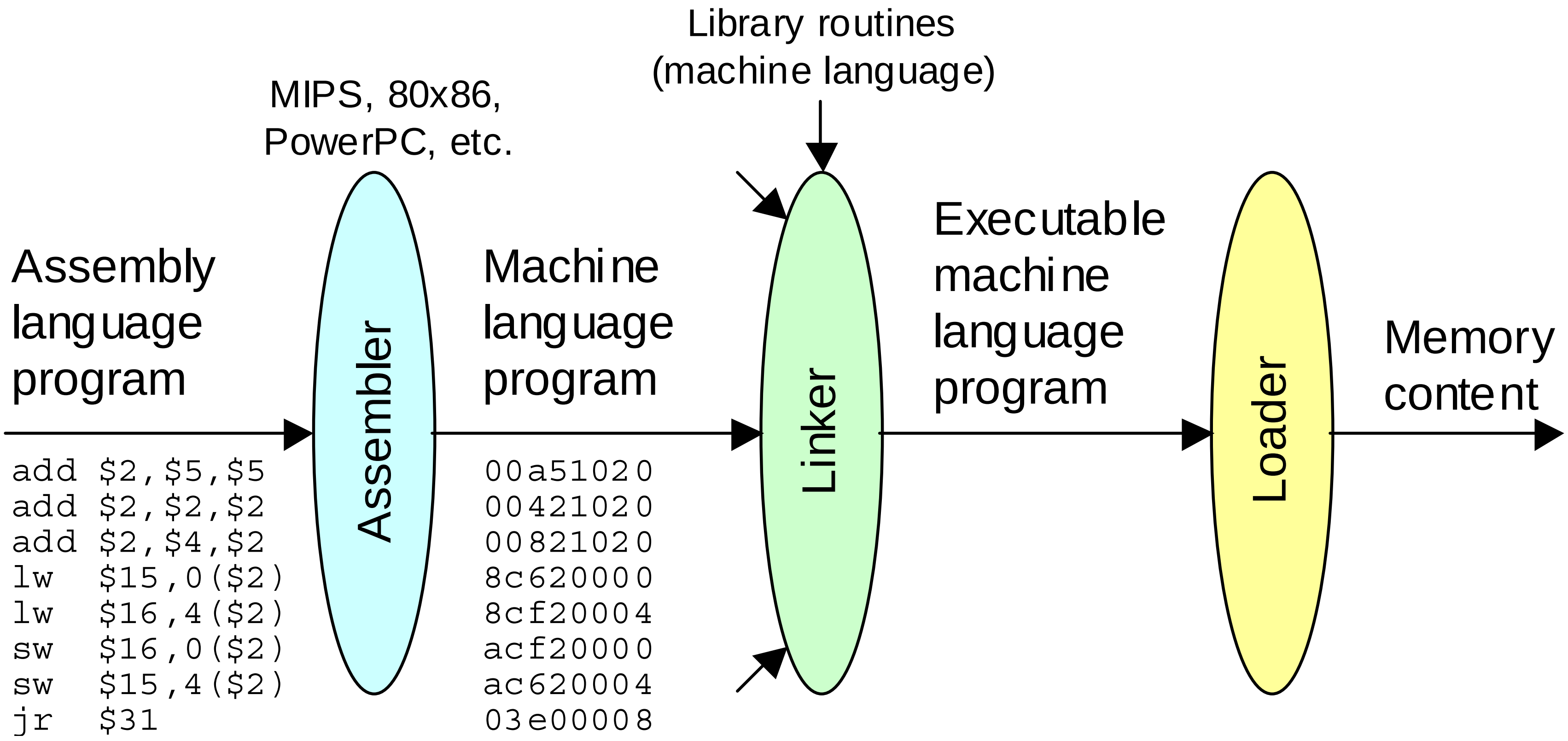
7 Assembly Language Programs

Everything else needed to build and run assembly programs:

- Supply info to assembler about program and its data
- Non-hardware-supported instructions for convenience

Topics in This Chapter	
7.1	Machine and Assembly Languages
7.2	Assembler Directives
7.3	Pseudoinstructions
7.4	Macroinstructions
7.5	Linking and Loading
7.6	Running Assembler Programs

7.1 Machine and Assembly Languages



Library routines (machine language)

MIPS, 80x86, PowerPC, etc.

Assembly language program

Assembler

Machine language program

Linker

Executable machine language program

Loader

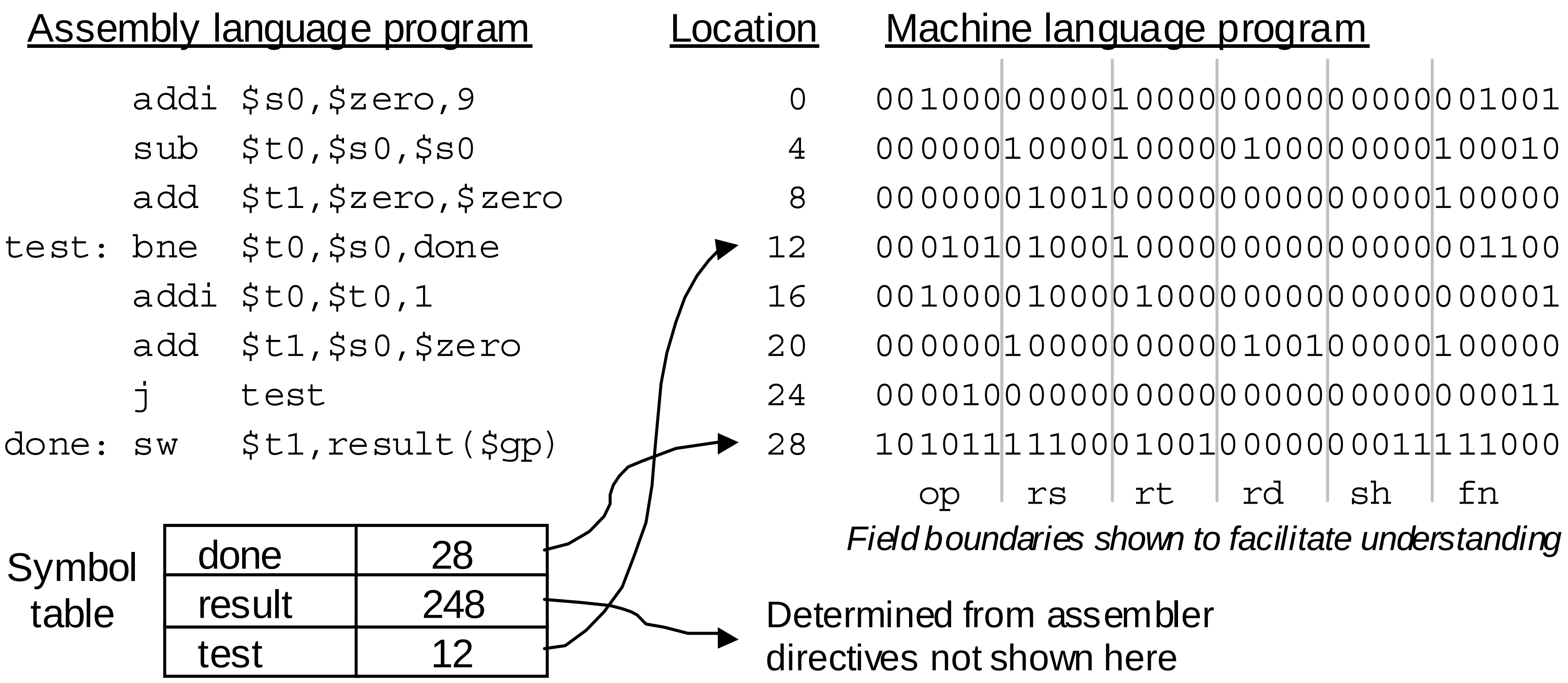
Memory content

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

```
00a51020
00421020
00821020
8c620000
8cf20004
acf20000
ac620004
03e00008
```

Figure 7.1 Steps in transforming an assembly language program to an executable program residing in memory.

Symbol Table



Assembly language program

```
      addi $s0,$zero,9
      sub  $t0,$s0,$s0
      add  $t1,$zero,$zero
test: bne  $t0,$s0,done
      addi $t0,$t0,1
      add  $t1,$s0,$zero
      j    test
done: sw   $t1,result($gp)
```

Location	Machine language program					
0	001000	00000	10000	00000	00000	001001
4	000000	10000	10000	01000	00000	100010
8	000000	01001	00000	00000	00000	100000
12	000101	01000	10000	00000	00000	001100
16	001000	01000	01000	00000	00000	000001
20	000000	10000	00000	01001	00000	100000
24	000010	00000	00000	00000	00000	000011
28	101011	11100	01001	00000	00011	111000
	op	rs	rt	rd	sh	fn

Field boundaries shown to facilitate understanding

Symbol table

done	28
result	248
test	12

Determined from assembler directives not shown here

Figure 7.2 An assembly-language program, its machine-language version, and the symbol table created during the assembly process.

7.2 Assembler Directives

Assembler directives provide the assembler with info on how to translate the program but do not lead to the generation of machine instructions

```
       .macro                  # start macro (see Section 7.4)
       .end_macro              # end macro (see Section 7.4)
       .text                   # start program’s text segment
       ...                     # program text goes here
       .data                   # start program’s data segment
 tiny: .byte    156,0x7a       # name & initialize data byte(s)
  max: .word    35000          # name & initialize data word(s)
small: .float   2E-3           # name short float (see Chapter 12)
  big: .double  2E-3           # name long float (see Chapter 12)
       .align   2              # align next item on word boundary
array: .space   600            # reserve 600 bytes = 150 words
 str1: .ascii   “a*b”          # name & initialize ASCII string
 str2: .asciiz  “xyz”          # null-terminated ASCII string
       .global  main           # consider “main” a global name
```

Composing Simple Assembler Directives

Example 7.1

Write assembler directive to achieve each of the following objectives:

a. Put the error message “Warning: The printer is out of paper!” in memory.
b. Set up a constant called “size” with the value 4.
c. Set up an integer variable called “width” and initialize it to 4.
d. Set up a constant called “mill” with the value 1,000,000 (one million).
e. Reserve space for an integer vector “vect” of length 250.

Solution:

```
a. noppr: .asciiz “Warning: The printer is out of paper!”
b.  size: .byte  4         # small constant fits in one byte
c. width: .word  4         # byte could be enough, but ...
d.  mill: .word  1000000   # constant too large for byte
e.  vect: .space 1000      # 250 words = 1000 bytes
```

7.3 Pseudoinstructions

Example of one-to-one pseudoinstruction: The following

```
not  $s0                # complement ($s0)
```

is converted to the real instruction:

```
nor  $s0,$s0,$zero      # complement ($s0)
```

Example of one-to-several pseudoinstruction: The following

```
abs  $t0,$s0            # put |($s0)| into $t0
```

is converted to the sequence of real instructions:

```
add  $t0,$s0,$zero      # copy x into $t0
slt  $at,$t0,$zero      # is x negative?
beq  $at,$zero,+4       # if not, skip next instr
sub  $t0,$zero,$s0      # the result is 0 – x
```

MiniMIPS Pseudo-instructions

Table 7.1

	Pseudoinstruction	Usage
Copy	Move	`move regd,regs`
	Load address	`la regd,address`
	Load immediate	`li regd,anyimm`
Arithmetic	Absolute value	`abs regd,regs`
	Negate	`neg regd,regs`
	Multiply (into register)	`mul regd,reg1,reg2`
	Divide (into register)	`div regd,reg1,reg2`
	Remainder	`rem regd,reg1,reg2`
	Set greater than	`sgt regd,reg1,reg2`
	Set less or equal	`sle regd,reg1,reg2`
	Set greater or equal	`sge regd,reg1,reg2`
Shift	Rotate left	`rol regd,reg1,reg2`
	Rotate right	`ror regd,reg1,reg2`
Logic	NOT	`not reg`
Memory access	Load doubleword	`ld regd,address`
	Store doubleword	`sd regd,address`
Control transfer	Branch less than	`blt reg1,reg2,L`
	Branch greater than	`bgt reg1,reg2,L`
	Branch less or equal	`ble reg1,reg2,L`
	Branch greater or equal	`bge reg1,reg2,L`

7.4 Macroinstructions

A macro is a mechanism to give a name to an often-used sequence of instructions (shorthand notation)

```
.macro  name(args)  # macro and arguments named
...                 # instr’s defining the macro
.end_macro          # macro terminator
```

How is a macro different from a pseudoinstruction?

Pseudos are predefined, fixed, and look like machine instructions
Macros are user-defined and resemble procedures (have arguments)

How is a macro different from a procedure?

Control is transferred to and returns from a procedure
After a macro has been replaced, no trace of it remains

Macro to Find the Largest of Three Values

Example 7.4

Write a macro to determine the largest of three values in registers and to put the result in a fourth register.

Solution:

```
.macro mx3r(m,a1,a2,a3)  # macro and arguments named
move   m,a1              # assume (a1) is largest; m = (a1)
bge    m,a2,+4           # if (a2) is not larger, ignore it
move   m,a2              # else set m = (a2)
bge    m,a3,+4           # if (a3) is not larger, ignore it
move   m,a3              # else set m = (a3)
.endmacro                # macro terminator
```

If the macro is used as `mx3r($t0,$s0,$s4,$s3)`, the assembler replaces the arguments `m`, `a1`, `a2`, `a3` with `$t0`, `$s0`, `$s4`, `$s3`, respectively.

7.5 Linking and Loading

The linker has the following responsibilities:

Ensuring correct interpretation (resolution) of labels in all modules
Determining the placement of text and data segments in memory
Evaluating all data addresses and instruction labels
Forming an executable program with no unresolved references

The loader is in charge of the following:

Determining the memory needs of the program from its header
Copying text and data from the executable program file into memory
Modifying (shifting) addresses, where needed, during copying
Placing program parameters onto the stack (as in a procedure call)
Initializing all machine registers, including the stack pointer
Jumping to a start-up routine that calls the program's main routine

7.6 Running Assembler Programs

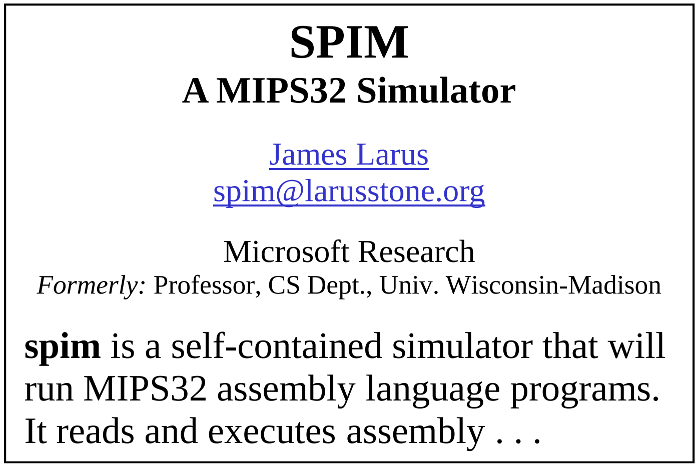
Spim is a simulator that can run MiniMIPS programs

The name Spim comes from reversing MIPS

Three versions of Spim are available for free downloading:

~~PCSpim~~	~~for Windows machines~~
~~xspim~~	~~for X-windows~~
~~spim~~	~~for Unix systems~~

QtSPIM for many OSs



SPIM
A MIPS32 Simulator

James Larus
spim@larusstone.org

Microsoft Research
Formerly: Professor, CS Dept., Univ. Wisconsin-Madison

spim is a self-contained simulator that will run MIPS32 assembly language programs. It reads and executes assembly . . .

You can download SPIM from:

http://www.cs.wisc.edu/~larus/spim.html

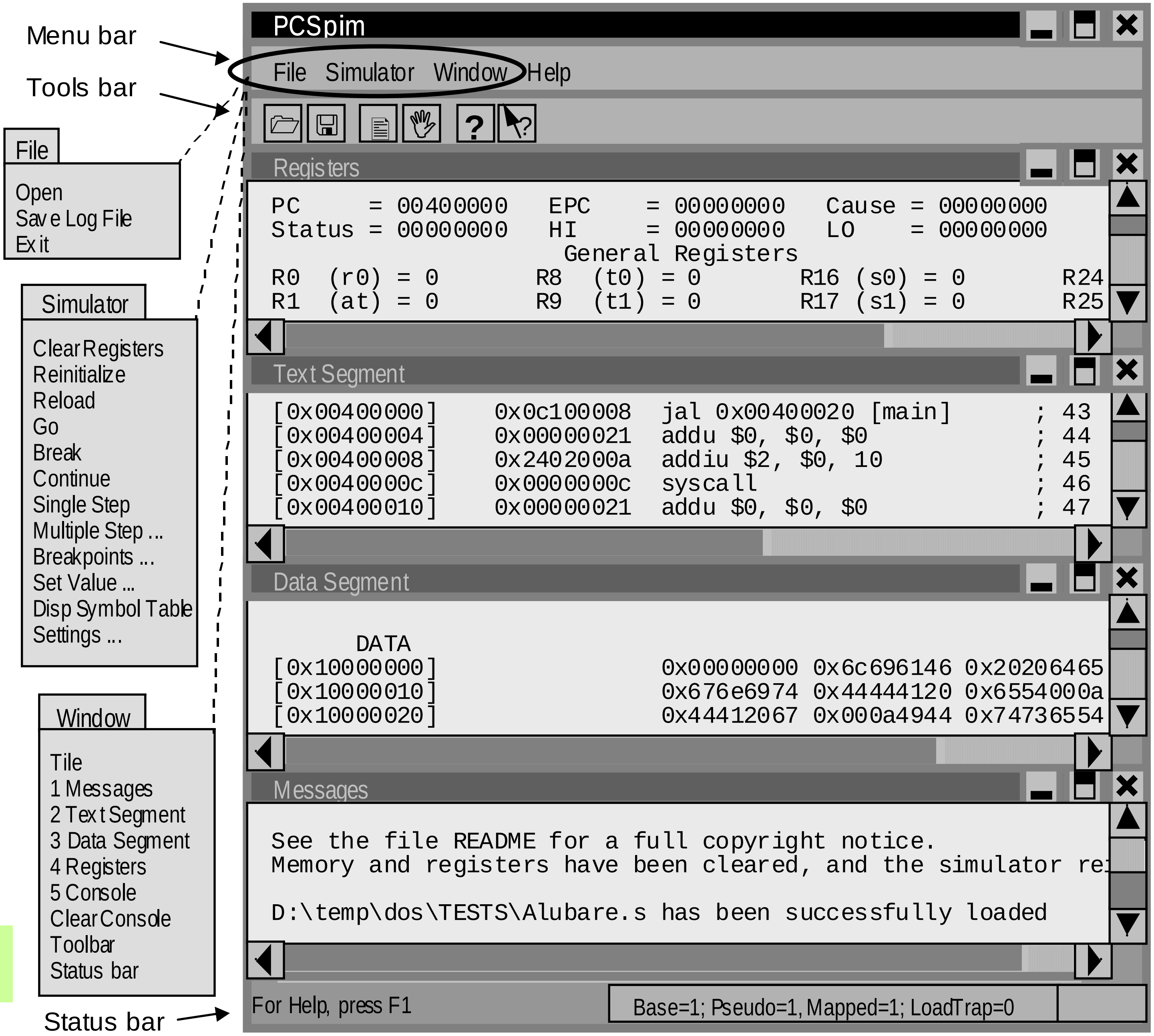
http://spimsimulator.sourceforge.net

Input/Output Conventions for MiniMIPS

Table 7.2 Input/output and control functions of `syscall` in PCSpim.

	($v0)	Function	Arguments	Result
Output	1	Print integer	Integer in `$a0`	Integer displayed
	2	Print floating-point	Float in `$f12`	Float displayed
	3	Print double-float	Double-float in `$f12`,`$f13`	Double-float displayed
	4	Print string	Pointer in `$a0`	Null-terminated string displayed
Input	5	Read integer		Integer returned in `$v0`
	6	Read floating-point		Float returned in `$f0`
	7	Read double-float		Double-float returned in `$f0`,`$f1`
	8	Read string	Pointer in `$a0`, length in `$a1`	String returned in buffer at pointer
Cntl	9	Allocate memory	Number of bytes in `$a0`	Pointer to memory block in `$v0`
	10	Exit from program		Program execution terminated

PCSpim User Interface



```
PC     = 00400000   EPC    = 00000000   Cause = 00000000
Status = 00000000   HI     = 00000000   LO    = 00000000
                      General Registers
R0  (r0) = 0        R8  (t0) = 0        R16 (s0) = 0        R24
R1  (at) = 0        R9  (t1) = 0        R17 (s1) = 0        R25
```

```
[0x00400000]    0x0c100008  jal 0x00400020 [main]      ; 43
[0x00400004]    0x00000021  addu $0, $0, $0            ; 44
[0x00400008]    0x2402000a  addiu $2, $0, 10           ; 45
[0x0040000c]    0x0000000c  syscall                    ; 46
[0x00400010]    0x00000021  addu $0, $0, $0            ; 47
```

```
      DATA
[0x10000000]                0x00000000 0x6c696146 0x20206465
[0x10000010]                0x676e6974 0x44444120 0x6554000a
[0x10000020]                0x44412067 0x000a4944 0x74736554
```

```
See the file README for a full copyright notice.
Memory and registers have been cleared, and the simulator re

D:\temp\dos\TESTS\Alubare.s has been successfully loaded
```

For Help, press F1   Base=1; Pseudo=1, Mapped=1; LoadTrap=0

Figure 7.3

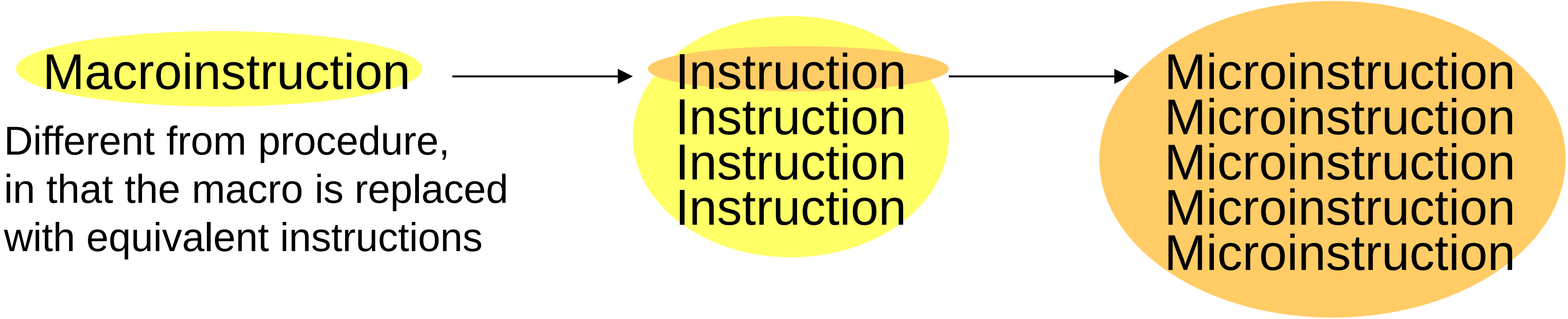
8 Instruction Set Variations

The MiniMIPS instruction set is only one example

- How instruction sets may differ from that of MiniMIPS
- RISC and CISC instruction set design philosophies

Topics in This Chapter
8.1 Complex Instructions
8.2 Alternative Addressing Modes
8.3 Variations in Instruction Formats
8.4 Instruction Set Design and Evolution
8.5 The RISC/CISC Dichotomy
8.6 Where to Draw the Line

Review of Some Key Concepts



Macroinstruction

Instruction
Instruction
Instruction
Instruction

Microinstruction
Microinstruction
Microinstruction
Microinstruction
Microinstruction

Different from procedure,
in that the macro is replaced
with equivalent instructions

Instruction format for a simple RISC design

	op	rs	rt	rd	sh	fn
	31	25	20	15	10	5 0
R	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
	Opcode	Source register 1	Source register 2	Destination register	Shift amount	Opcode extension

	op	rs	rt	operand / offset
	31	25	20	15 0
I	6 bits	5 bits	5 bits	16 bits
	Opcode	Source or base	Destination or data	Immediate operand or address offset

	op	jump target address
	31	25 0
J	6 bits	26 bits
	Opcode	Memory word address (byte address divided by 4)

All of the same length

Fields used consistently (simple decoding)

Can initiate reading of registers even before decoding the instruction

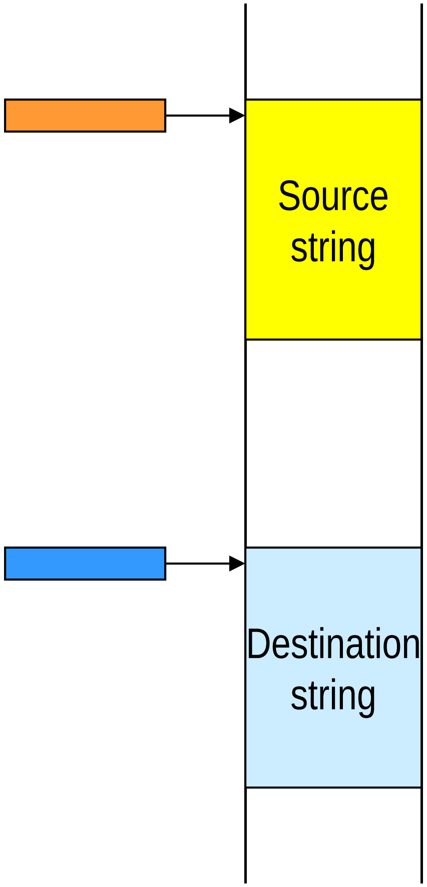
Short, uniform execution

8.1 Complex Instructions

Table 8.1 (partial) Examples of complex instructions in two popular modern microprocessors and two computer families of historical significance

Machine	Instruction	Effect
Pentium	`MOVS`	Move one element in a string of bytes, words, or doublewords using addresses specified in two pointer registers; after the operation, increment or decrement the registers to point to the next element of the string
PowerPC	`cntlzd`	Count the number of consecutive 0s in a specified source register beginning with bit position 0 and place the count in a destination register
IBM 360-370	`CS`	Compare and swap: Compare the content of a register to that of a memory location; if unequal, load the memory word into the register, else store the content of a different register into the same memory location
Digital VAX	`POLYD`	Polynomial evaluation with double flp arithmetic: Evaluate a polynomial in *x*, with very high precision in intermediate results, using a coefficient table whose location in memory is given within the instruction

Some Details of Sample Complex Instructions

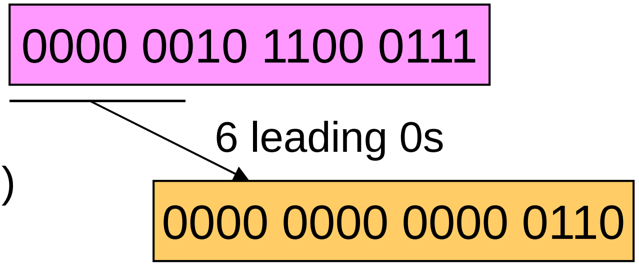


MOVS

(Move string)

cntlzd

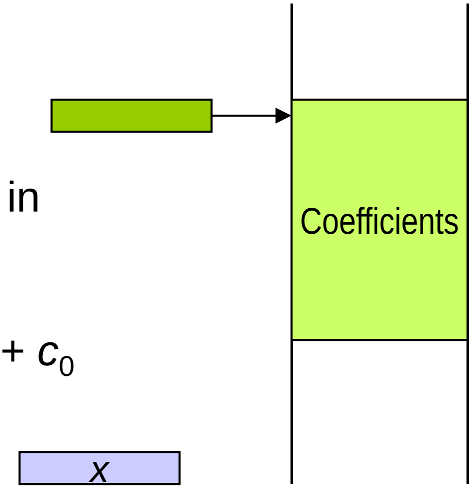
(Count leading 0s)



POLYD

(Polynomial evaluation in double floating-point)

$c_{n-1}x^{n-1} + \cdots + c_2x^2 + c_1x + c_0$



Benefits and Drawbacks of Complex Instructions

Fewer instructions in program
(less memory)

Fewer memory accesses for instructions

Programs may become easier to write/read/understand

Potentially faster execution
(complex steps are still done sequentially in multiple cycles, but hardware control can be faster than software loops)

More complex format
(slower decoding)

Less flexible
(one algorithm for polynomial evaluation or sorting may not be the best in all cases)

If interrupts are processed at the end of instruction cycle, machine may become less responsive to time-critical events (interrupt handling)

8.2 Alternative Addressing Modes

Let's refresh our memory (from Chap. 5)

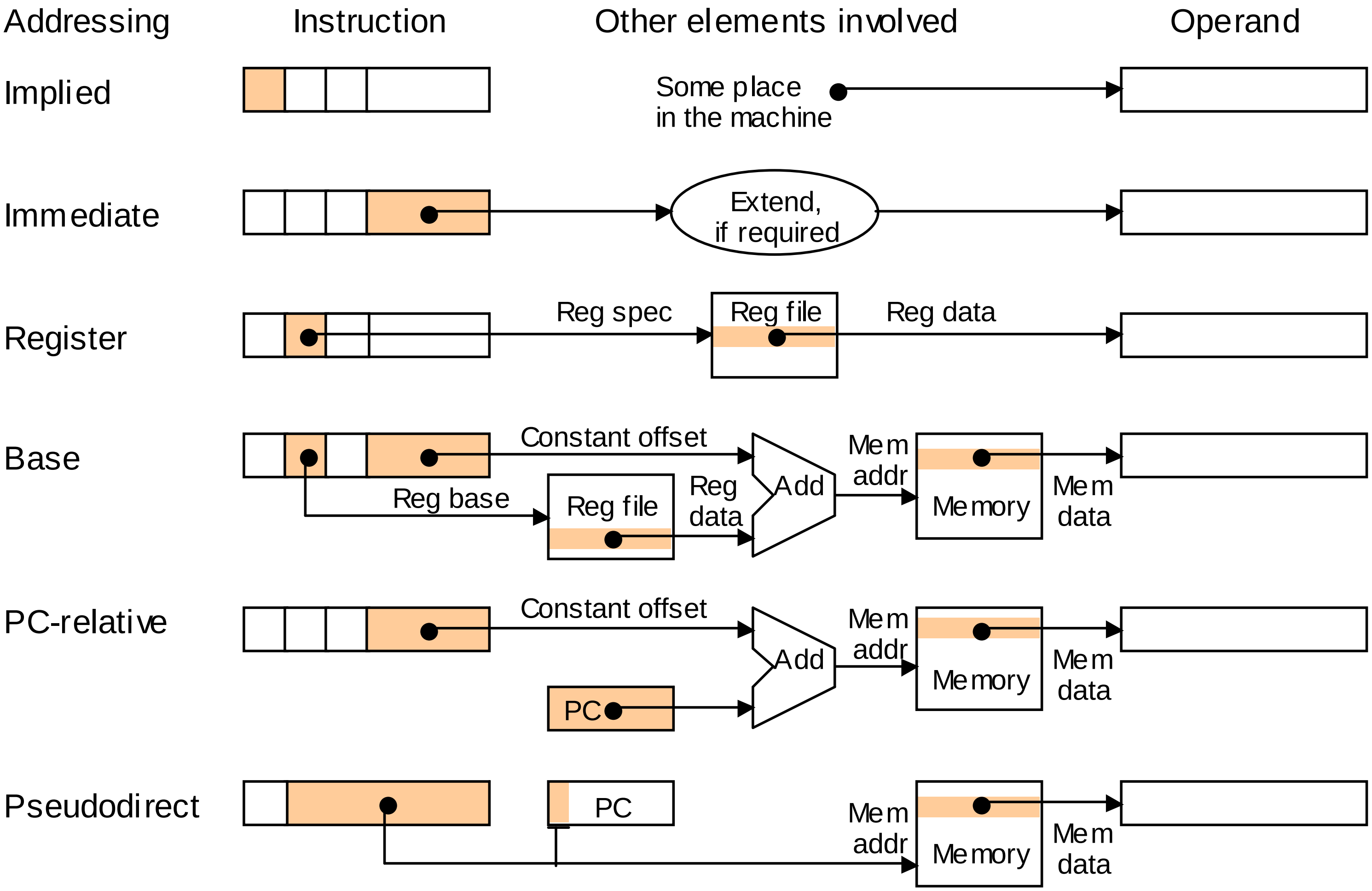


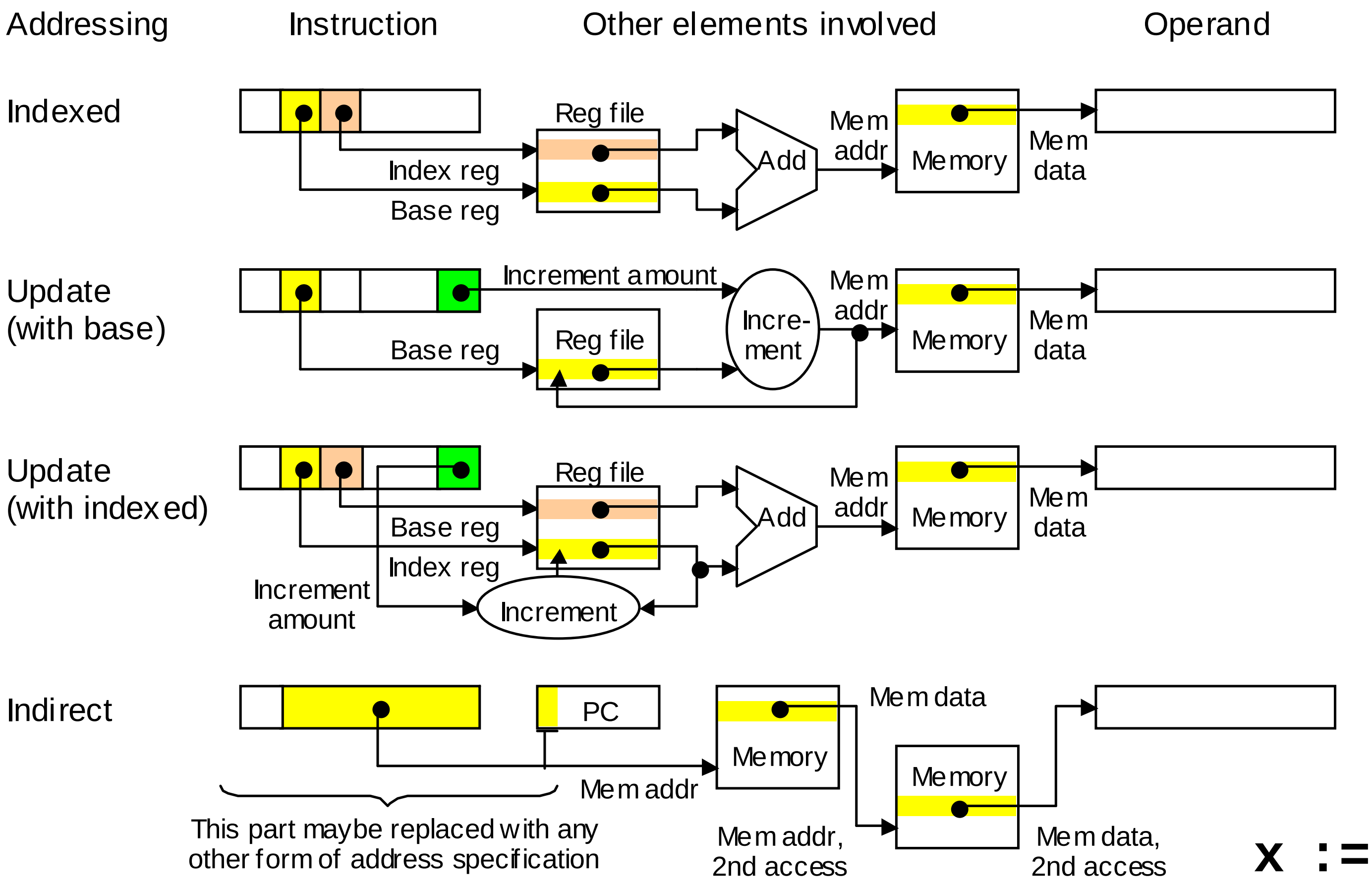
Figure 5.11 Schematic representation of addressing modes in MiniMIPS.

Table 6.2 Addressing Mode Examples in the MiniMIPS ISA

Instruction	Usage
Load upper immediate	`lui rt,imm`
Add	`add rd,rs,rt`
Subtract	`sub rd,rs,rt`
Set less than	`slt rd,rs,rt`
Add immediate	`addi rt,rs,imm`
Set less than immediate	`slti rd,rs,imm`
AND	`and rd,rs,rt`
OR	`or rd,rs,rt`
XOR	`xor rd,rs,rt`
NOR	`nor rd,rs,rt`
AND immediate	`andi rt,rs,imm`
OR immediate	`ori rt,rs,imm`
XOR immediate	`xori rt,rs,imm`
Load word	`lw rt,imm(rs)`
Store word	`sw rt,imm(rs)`
Jump	`j L`
Jump register	`jr rs`
Branch less than 0	`bltz rs,L`
Branch equal	`beq rs,rt,L`
Branch not equal	`bne rs,rt,L`

Instruction	Usage
Move from Hi	`mfhi rd`
Move from Lo	`mflo rd`
Add unsigned	`addu rd,rs,rt`
Subtract unsigned	`subu rd,rs,rt`
Multiply	`mult rs,rt`
Multiply unsigned	`multu rs,rt`
Divide	`div rs,rt`
Divide unsigned	`divu rs,rt`
Add immediate unsigned	`addiu rs,rt,imm`
Shift left logical	`sll rd,rt,sh`
Shift right logical	`srl rd,rt,sh`
Shift right arithmetic	`sra rd,rt,sh`
Shift left logical variable	`sllv rd,rt,rs`
Shift right logical variable	`srlv rd,rt,rs`
Shift right arith variable	`srav rd,rt,rs`
Load byte	`lb rt,imm(rs)`
Load byte unsigned	`lbu rt,imm(rs)`
Store byte	`sb rt,imm(rs)`
Jump and link	`jal L`
System call	`syscall`

More Elaborate Addressing Modes



```
x := B[i]

x := Mem[p]
p := p + 1

x := B[i]
i := i + 1

t := Mem[p]
x := Mem[t]

x := Mem[Mem[p]]
```

Figure 8.1 Schematic representation of more elaborate addressing modes not supported in MiniMIPS.

Usefulness of Some Elaborate Addressing Modes

Update mode: XORing a string of bytes

```
loop: lb   $t0,A($s0)
      xor  $s1,$s1,$t0
      addi $s0,$s0,-1
      bne  $s0,$zero,loop
```

One instruction with update addressing

Indirect mode: Case statement

```
case: lw   $t0,0($s0)  # get s
      add  $t0,$t0,$t0 # form 2s
      add  $t0,$t0,$t0 # form 4s
      la   $t1,T       # base T
      add  $t1,$t0,$t1
      lw   $t2,0($t1)  # entry
      jr   $t2
```

Branch to location L*i* if s = *i* (switch var.)

T	L0
T+4	L1
T+8	L2
T+12	L3
T+16	L4
T+20	L5

8.3 Variations in Instruction Formats

0-, 1-, 2-, and 3-address instructions in MiniMIPS

Category	Format	Opcode	Description of operand(s)
0-address	0 ▨ 12	`syscall`	One implied operand in register `$v0`
1-address	2 Address	`j`	Jump target addressed (in pseudodirect form)
2-address	0 rs rt ▨ ▨ 24	`mult`	Two source registers addressed, destination implied
3-address	0 rs rt rd ▨ 32	`add`	Destination and two source registers addressed

Figure 8.2 Examples of MiniMIPS instructions with 0 to 3 addresses; shaded fields are unused.

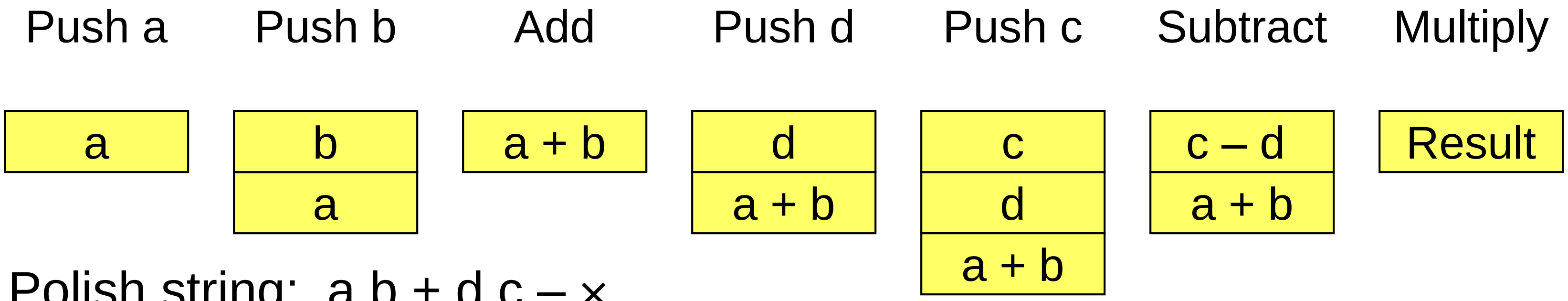
Zero-Address Architecture: Stack Machine

Stack holds all the operands (replaces our register file)

Load/Store operations become push/pop

Arithmetic/logic operations need only an opcode: they pop operand(s) from the top of the stack and push the result onto the stack

Example: Evaluating the expression (a + b) × (c – d)



Push a	Push b	Add	Push d	Push c	Subtract	Multiply
a	b	a + b	d	c	c – d	Result
	a		a + b	d	a + b	
				a + b		

Polish string: a b + d c – ×

If a variable is used again, you may have to push it multiple times

Special instructions such as “Duplicate” and “Swap” are helpful

One-Address Architecture: Accumulator Machine

The accumulator, a special register attached to the ALU, always holds operand 1 and the operation result

Only one operand needs to be specified by the instruction

Example: Evaluating the expression (a + b) × (c – d)

```
Load      a
add       b
Store     t
load      c
subtract  d
multiply  t
```

Within branch instructions, the condition or target address must be implied

Branch to L if acc negative

If register x is negative skip the next instruction

May have to store accumulator contents in memory (example above)

No store needed for a + b + c + d + . . . (“accumulator”)

Two-Address Architectures

Two addresses may be used in different ways:

Operand1/result and operand 2

Condition to be checked and branch target address

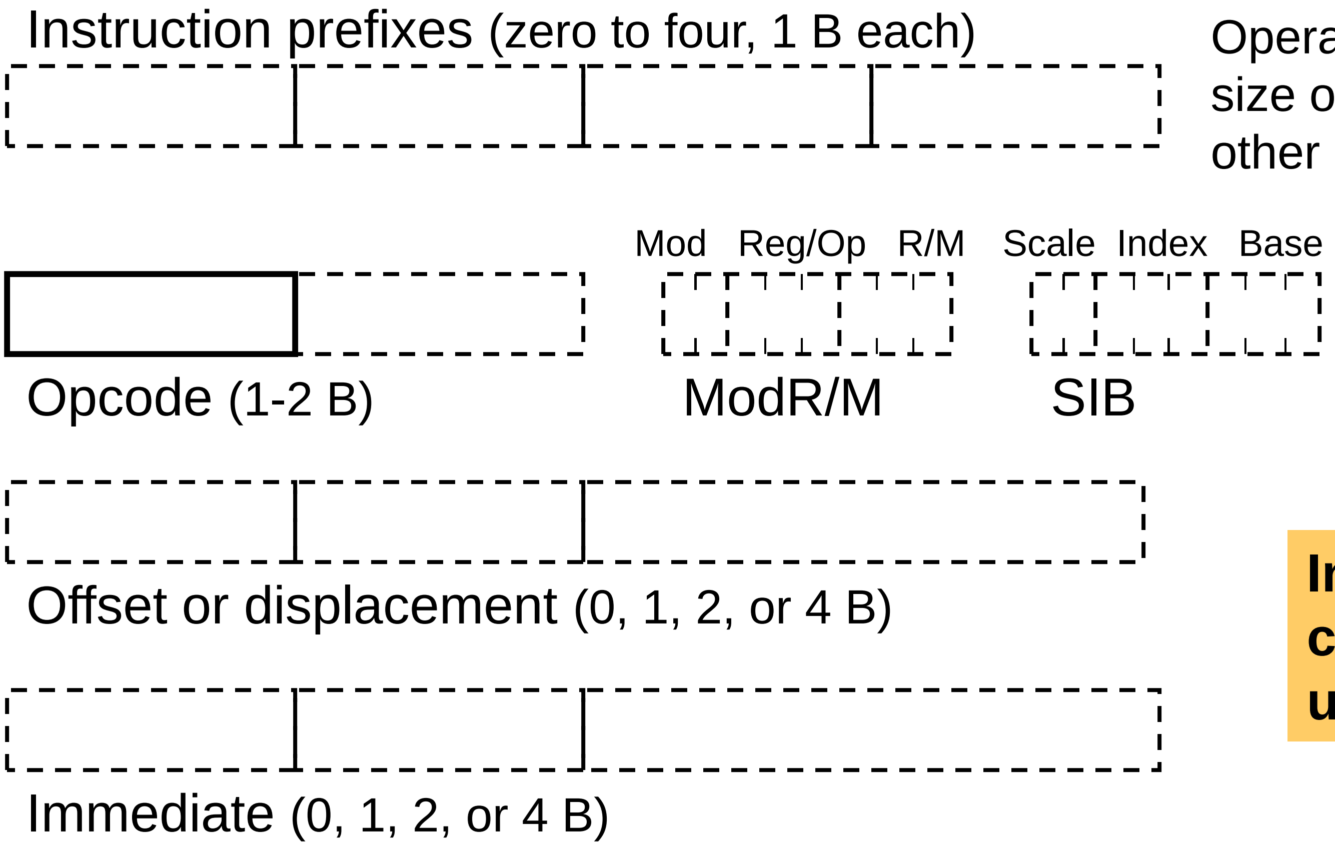
Example: Evaluating the expression (a + b) × (c – d)

```
load     $1,a
add      $1,b
load     $2,c
subtract $2,d
multiply $1,$2
```

Instructions of a hypothetical two-address machine

A variation is to use one of the addresses as in a one-address machine and the second one to specify a branch in every instruction

Example of a Complex Instruction Format



Instruction prefixes (zero to four, 1 B each)

Operand/address size overwrites and other modifiers

Mod Reg/Op R/M

Scale Index Base

Opcode (1-2 B)

ModR/M

SIB

Most memory operands need these 2 bytes

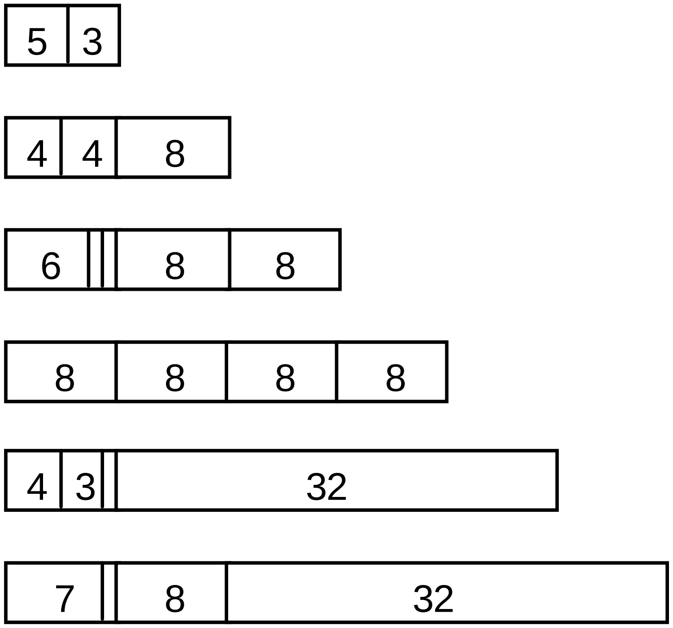
Offset or displacement (0, 1, 2, or 4 B)

Immediate (0, 1, 2, or 4 B)

Instructions can contain up to 15 bytes

Components that form a variable-length IA-32 (80x86) instruction.

Some of IA-32’s Variable-Width Instructions



Type	Format (field widths shown)	Opcode	Description of operand(s)
1-byte	5 3	`PUSH`	3-bit register specification
2-byte	4 4 8	`JE`	4-bit condition, 8-bit jump offset
3-byte	6 8 8	`MOV`	8-bit register/mode, 8-bit offset
4-byte	8 8 8 8	`XOR`	8-bit register/mode, 8-bit base/index, 8-bit offset
5-byte	4 3 32	`ADD`	3-bit register spec, 32-bit immediate
6-byte	7 8 32	`TEST`	8-bit register/mode, 32-bit immediate

Figure 8.3 Example 80x86 instructions ranging in width from 1 to 6 bytes; much wider instructions (up to 15 bytes) also exist

8.4 Instruction Set Design and Evolution

Desirable attributes of an instruction set:

Consistent, with uniform and generally applicable rules
Orthogonal, with independent features noninterfering
Transparent, with no visible side effect due to implementation details
Easy to learn/use (often a byproduct of the three attributes above)
Extensible, so as to allow the addition of future capabilities
Efficient, in terms of both memory needs and hardware realization

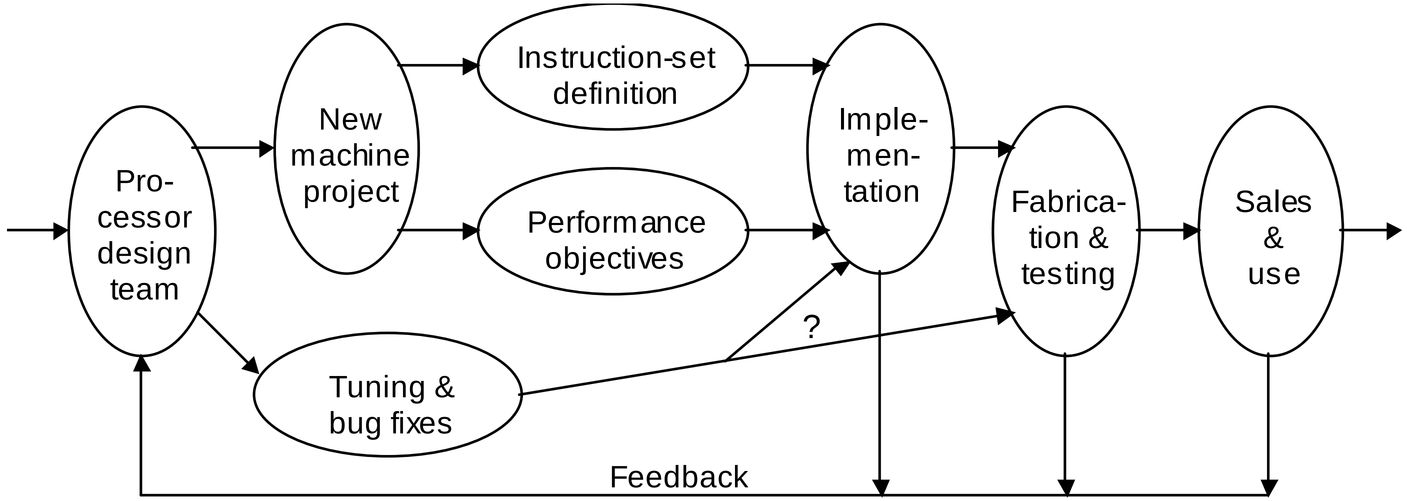


Figure 8.4 Processor design and implementation process.

8.5 The RISC/CISC Dichotomy

The RISC (reduced instruction set computer) philosophy: Complex instruction sets are undesirable because inclusion of mechanisms to interpret all the possible combinations of opcodes and operands might slow down even very simple operations.

Ad hoc extension of instruction sets, while maintaining backward compatibility, leads to CISC; imagine modern English containing every English word that has been used through the ages

Features of RISC architecture

1. Small set of inst's, each executable in roughly the same time
2. Load/store architecture (leading to more registers)
3. Limited addressing mode to simplify address calculations
4. Simple, uniform instruction formats (ease of decoding)

RISC/CISC Comparison via Generalized Amdahl’s Law

Example 8.1

An ISA has two classes of simple (S) and complex (C) instructions. On a reference implementation of the ISA, class-S instructions account for 95% of the running time for programs of interest. A RISC version of the machine is being considered that executes only class-S instructions directly in hardware, with class-C instructions treated as pseudoinstructions. It is estimated that in the RISC version, class-S instructions will run 20% faster while class-C instructions will be slowed down by a factor of 3. Does the RISC approach offer better or worse performance compared to the reference implementation?

Solution

Per assumptions, 0.95 of the work is speeded up by a factor of 1.0 / 0.8 = 1.25, while the remaining 5% is slowed down by a factor of 3. The RISC speedup is 1 / [0.95 / 1.25 + 0.05 × 3] = 1.1. Thus, a 10% improvement in performance can be expected in the RISC version.

Some Hidden Benefits of RISC

In Example 8.1, we established that a speedup factor of 1.1 can be expected from the RISC version of a hypothetical machine

This is not the entire story, however!

If the speedup of 1.1 came with some additional cost, then one might legitimately wonder whether it is worth the expense and design effort

The RISC version of the architecture also:

Reduces the effort and team size for design

Shortens the testing and debugging phase

Simplifies documentation and maintenance

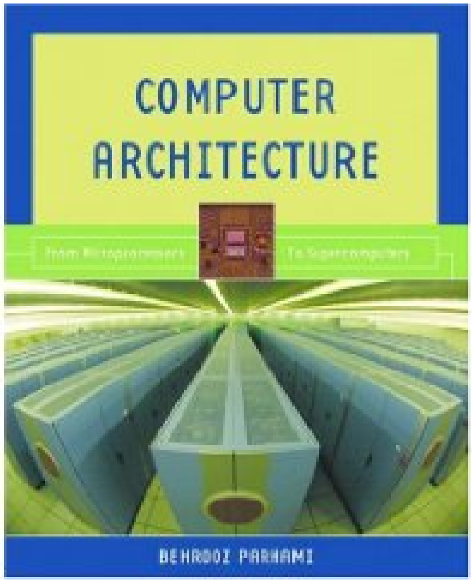
Cheaper product and shorter time-to-market

MIPS Performance Rating Revisited

An *m*-MIPS processor can execute *m* million instructions per second

Comparing an *m*-MIPS processor with a 10*m*-MIPS processor
Like comparing two people who read *m* pages and 10*m* pages per hour

10 pages / hr



100 pages / hr

Reading 100 pages per hour, as opposed to 10 pages per hour, may not allow you to finish the same reading assignment in 1/10 the time

RISC / CISC Convergence

The earliest RISC designs:
CDC 6600, highly innovative supercomputer of the mid 1960s
IBM 801, influential single-chip processor project of the late 1970s

In the early 1980s, two projects brought RISC to the forefront:
UC Berkeley's RISC 1 and 2, forerunners of the Sun SPARC
Stanford's MIPS, later marketed by a company of the same name

Throughout the 1980s, there were heated debates about the relative merits of RISC and CISC architectures

Since the 1990s, the debate has cooled down!

We can now enjoy both sets of benefits by having complex instructions automatically translated to sequences of very simple instructions that are then executed on RISC-based underlying hardware

8.6 Where to Draw the Line

The ultimate reduced instruction set computer (URISC):

How many instructions are absolutely needed for useful computation?

Only one!

```
subtract source1 from source2, replace source2 with the
result, and jump to target address if result is negative
```

Assembly language form:

```
label: urisc  dest,src1,target
```

Pseudoinstructions can be synthesized using the single instruction:

```
 stop: .word  0
start: urisc  dest,dest,+1  # dest = 0
       urisc  temp,temp,+1  # temp = 0
       urisc  temp,src,+1   # temp = -(src)
       urisc  dest,temp,+1  # dest = -(temp); i.e. (src)
       ...                  # rest of program
```

This is the `move` pseudoinstruction

Corrected version

Some Useful Pseudo Instructions for URISC

Example 8.2 (2 parts of 5)

Write the sequence of instructions that are produced by the URISC assembler for each of the following pseudoinstructions.

```
parta: uadd  dest,src1,src2  # dest=(src1)+(src2)
partc: uj    label           # goto label
```

Solution

`at1` and `at2` are temporary memory locations for assembler’s use

```
parta: urisc  at1,at1,+1      # at1 = 0
       urisc  at1,src1,+1     # at1 = -(src1)
       urisc  at1,src2,+1     # at1 = -(src1)–(src2)
       urisc  dest,dest,+1    # dest = 0
       urisc  dest,at1,+1     # dest = -(at1)
partc: urisc  at1,at1,+1      # at1 = 0
       urisc  at1,one,label   # at1 = -1 to force jump
```

URISC Hardware

URISC instruction:

Word 1	Word 2	Word 3
Source 1	Source 2 / Dest	Jump target

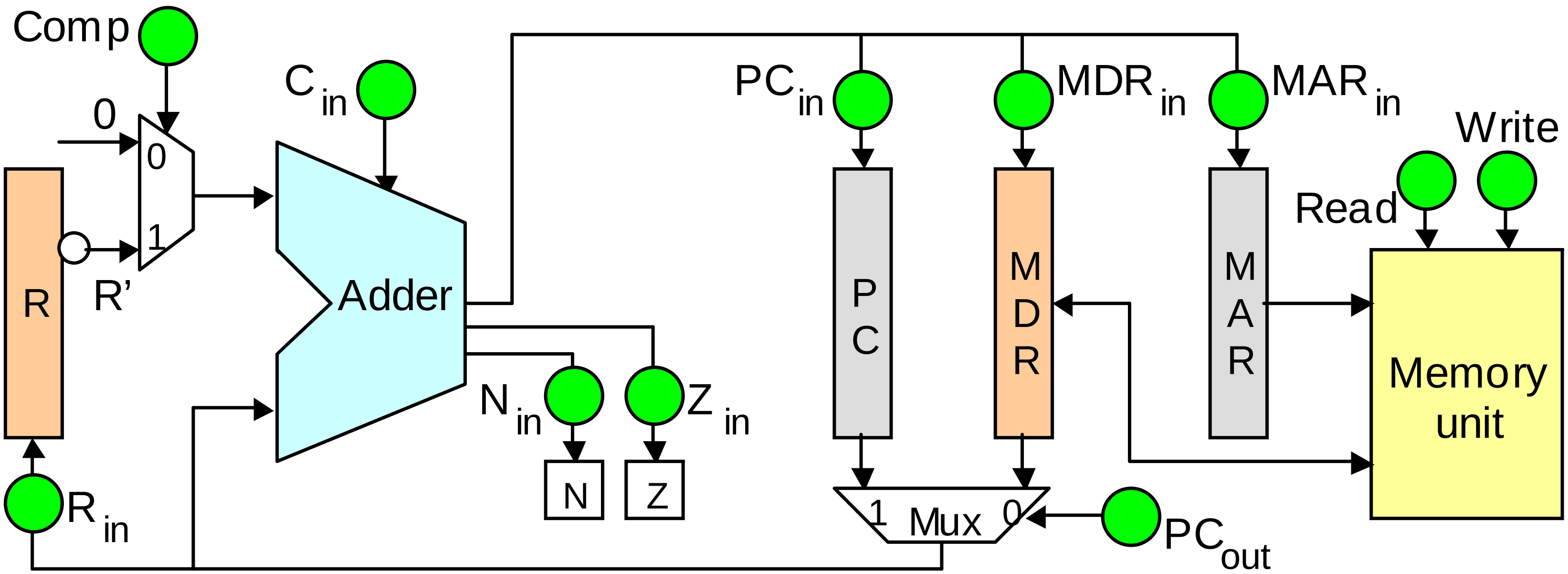


Figure 8.5 Instruction format and hardware structure for URISC.