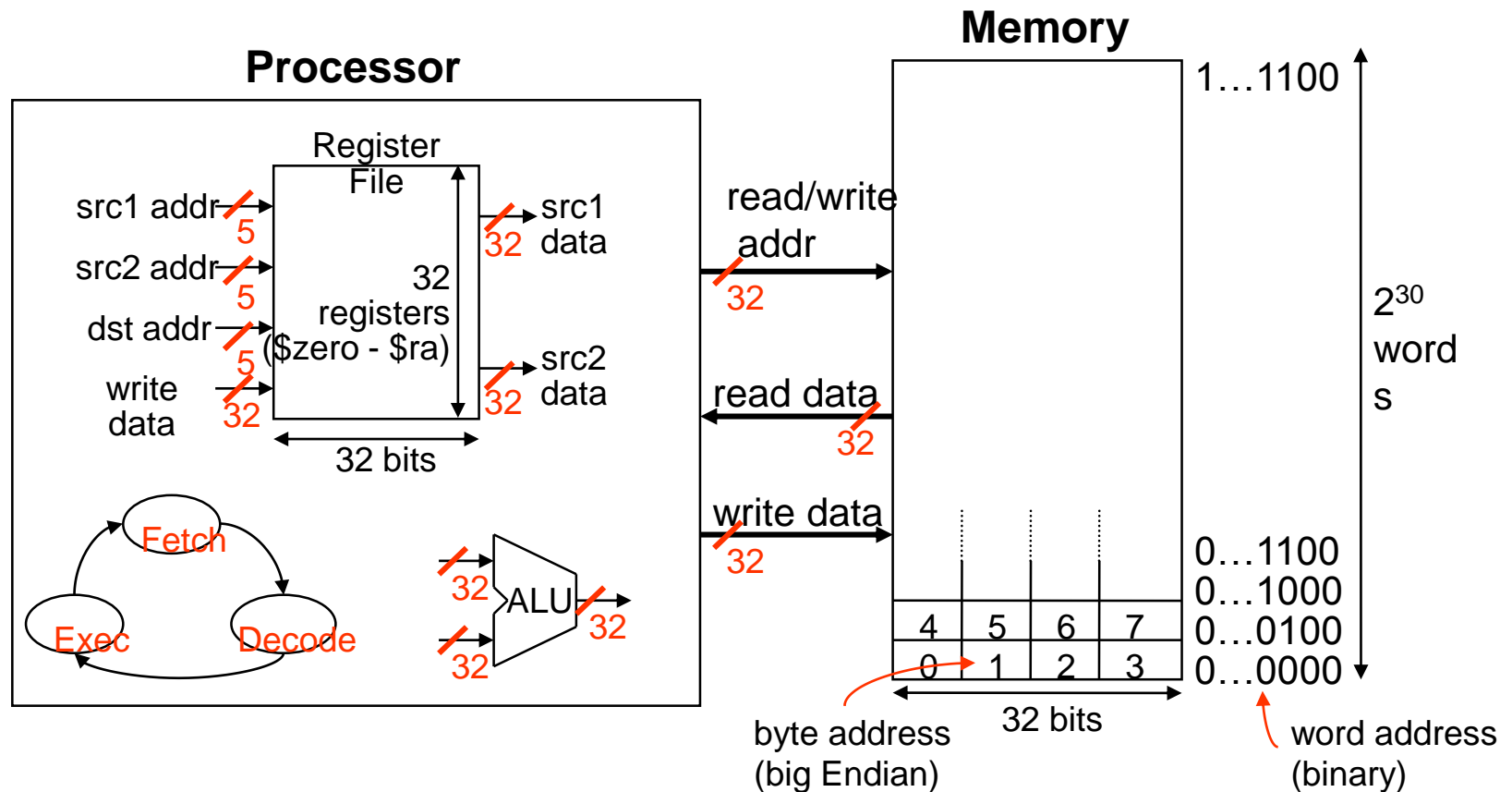
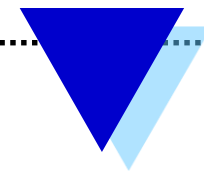


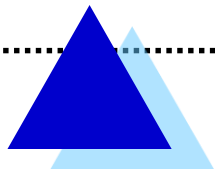
Machine Language Instructions



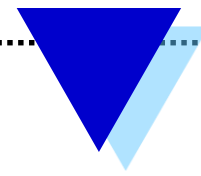
Instruction Set



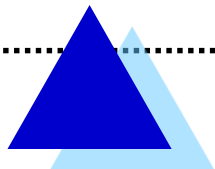
- **Instruction set** – the vocabulary which the computer understands. Language of the machine
- Assembly language is more primitive than higher level languages
e.g., no sophisticated control flow
- Goal: a language that makes it easy to build the hardware and compiler while maximizing performance and minimizing cost and power
- Many modern computers have simple instruction sets



Assembly Language



- Primarily the **MIPS** instruction set architecture
 - similar to other architectures – ARM and Intel x86
 - used by Intel, NEC, Nintendo, Silicon Graphics, Sony, ...
 - 32-bit architecture
 - 32 bit data line and address line
 - data and addresses are 32-bit
- Stanford MIPS commercialized by MIPS Technologies (www.mips.com)
- Used by many embedded systems (cameras, printers, consumer electronics)

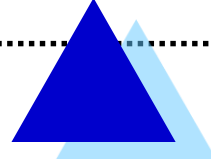


Assembly Language

- MIPS favors simplicity (enables higher performance at lower cost) – each line in the program is an instruction.
- There are several categories of instructions:
- Some are “register type” or **R-type** (such as arithmetic operations) and
- Some are “Data access type” or **I-type** (**lw**, **sw**) – involve memory access.
- All instructions are represented as **32-bit binary numbers**, but the significance of the bits differs depending on the instruction category (as we’ll see later).

Arithmetic instructions

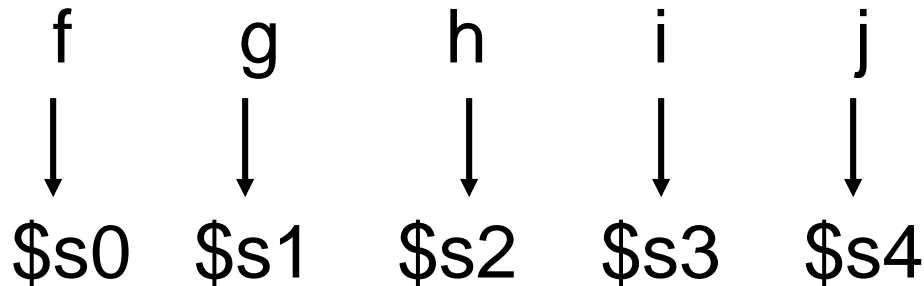


- MIPS assembly language arithmetic statement (like add a, b, c #a=b+c)
 - add \$t0, \$s1, \$s2
 - sub \$t0, \$s1, \$s2
 - Each arithmetic instruction performs only **one** operation
 - Each arithmetic instruction specifies exactly **three** operands
 - destination ← source1 op source2 op
 - Those operands are contained in the datapath's **register file** (\$t0, \$s1, \$s2)
 - Operand order is fixed (destination first)
- 

More complex instructions

◆ **Example** The C statement $f = (g+h) - (i+j);$

If variable values are stored in registers

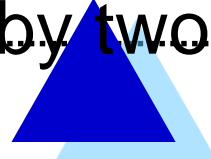


Then the same statement can be re-written in MIPS assembly as

```
add  $t0, $s1, $s2  # $t0 = g + h
add  $t1, $s3, $s4  # $t1 = i + j
sub  $s0, $t0, $t1   # $s0 = f
```

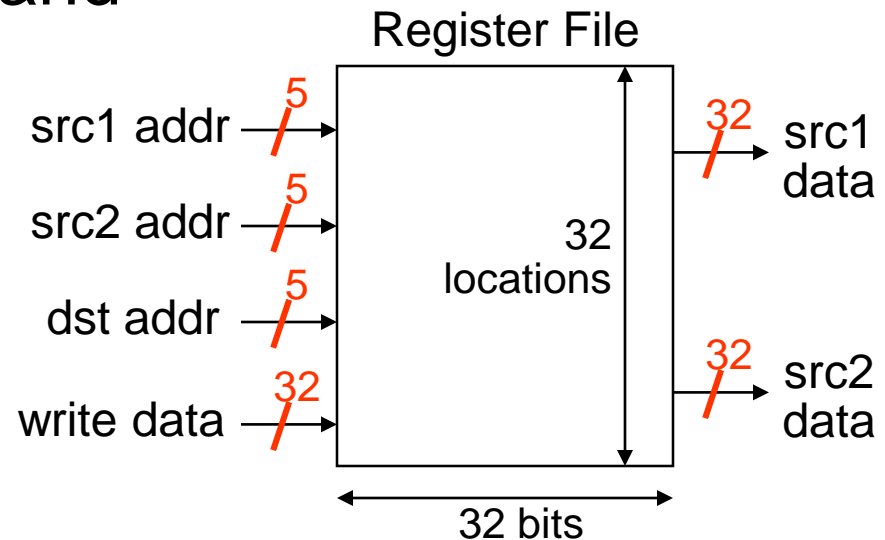
Operands and Registers



- Operands are stored in Registers (memory locations visible to the programmer)
 - Registers are
 - Faster than main memory
 - Can hold variables so that
 - code density improves (since registers are named with fewer bits than a memory location)
 - – why is that?
 - Smaller is faster (much faster than main memory)
 - Register addresses are indicated by using \$ followed by two characters (like \$t0)
- 

MIPS Register File

- Operands of arithmetic instructions must be from a limited number of special locations contained in the datapath's **register file** (inside the CPU)
 - Holds thirty-two 32-bit registers
 - With two read ports and
 - One write port

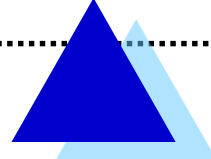




Name Convention for Registers

Registers 0.. 15

0	\$zero constant 0
1	\$at reserved for assembler
2	\$v0 expression evaluation &
3	\$v1 function results
4	\$a0 argument registers
5	\$a1
6	\$a2
7	\$a3
8	\$t0 temporary: caller saves
...	(callee can clobber)
15	\$t7



Name Convention for Registers

Registers 16... 31

16 **\$s0** callee saves

... 18 **\$s2**

23 **\$s7**

24 **\$t8** temporary (cont'd)

25 **\$t9**

26 **\$k0** reserved for OS kernel

27 **\$k1**

28 **\$gp** pointer to global area

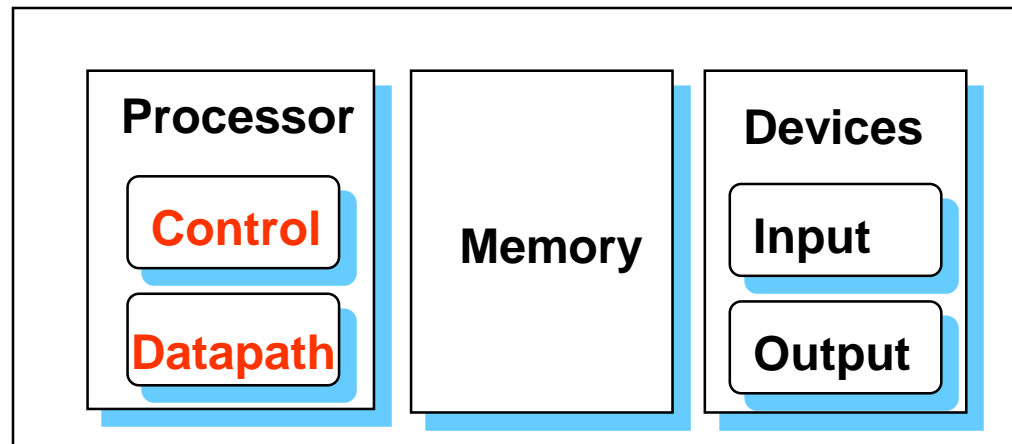
29 **\$sp** stack pointer

30 **\$fp** frame pointer

31 **\$ra** return address

What happens for large arrays?

- Arithmetic instructions operands must be registers,
— only thirty-two registers provided
- What about programs with *lots* of variables?
 - Store variables in memory
 - Load variables from memory to registers before use; store them back to memory after use.

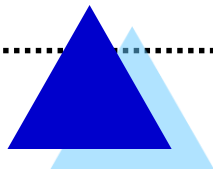




Accessing Memory

The data transfer instruction must specify

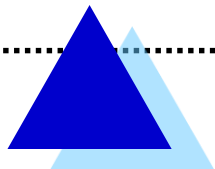
- where in memory to read from (load) or to write to (store) – **memory address**
- where in the register file to write to (load) or read from (store) – **register destination (source)**
- The memory address is formed by summing **the constant** portion of the instruction and **the contents of the second register in the same instruction**



Accessing Memory

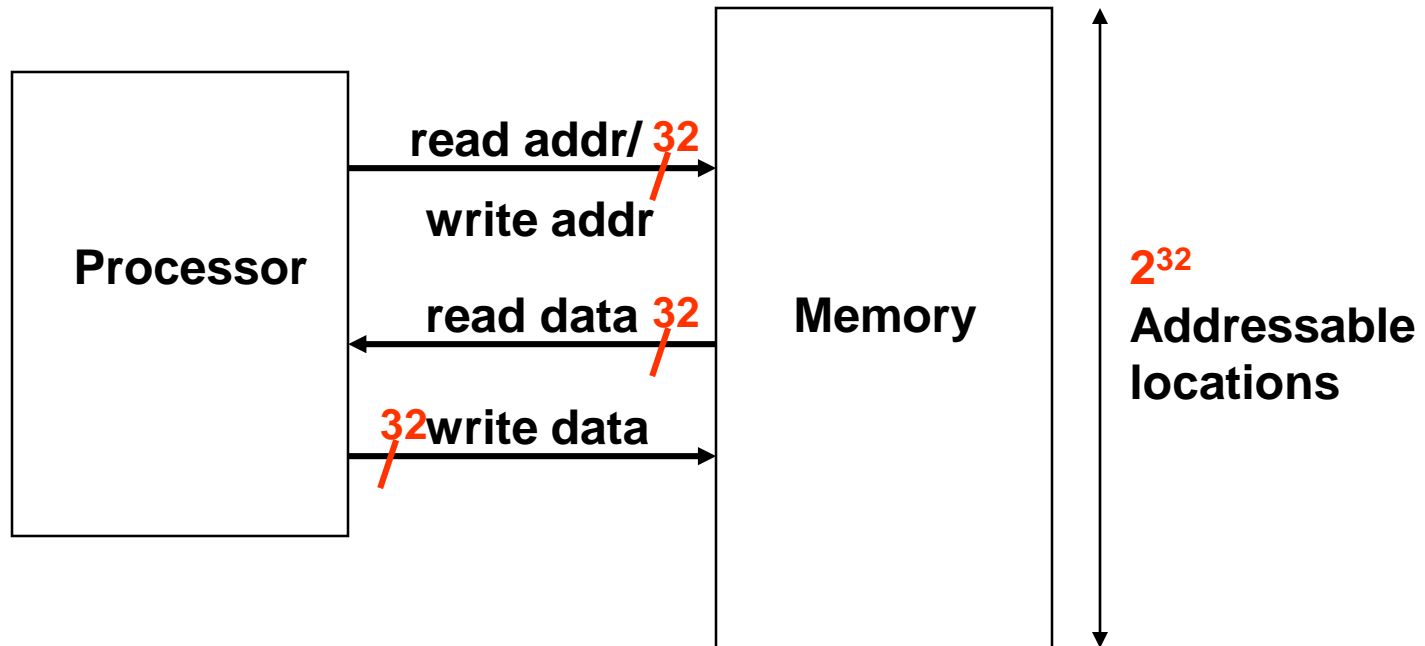


- MIPS has two basic **data transfer** instructions for accessing memory
- **lw** `$t0, 4($s3)` #load word –
read data from memory and place it in
register `$t0`
- **sw** `$t1, 8($s3)` #store word –
store data located in register `$t1` to
memory



Addressing Memory

- Memory is viewed as a large, single-dimension array, with an address
- A memory address is an *index* into the array



Addressing Memory

- Since 8-bit bytes are so useful, most architectures address individual *bytes* in memory
- With 32-bit addressing – we can access 2^{32} bytes
- Depending on how bytes are address-sequenced in a word, architectures may be “Big endian” (like MIPS) or “Little endian”

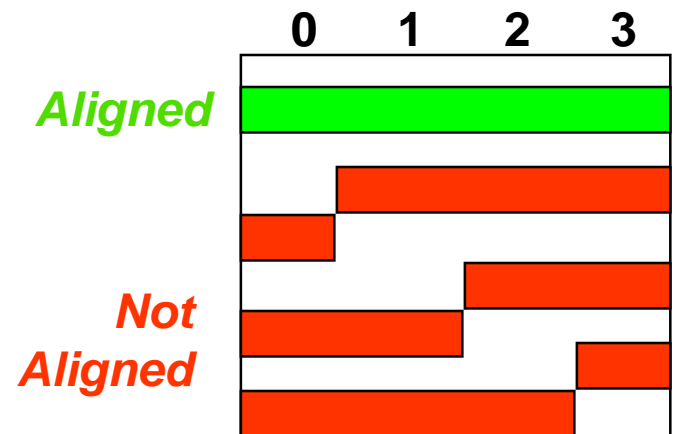
Byte #			
0	1	2	3

Big endian (MIPS)

Addressing Memory

- memory: 2^{32} bytes = 2^{30} words
- Therefore, the memory address of a **word** must be a multiple of 4 (**alignment restriction**) – leads to faster data transfers

Alignment restriction: requires that objects fall on an address that is multiple of their size.

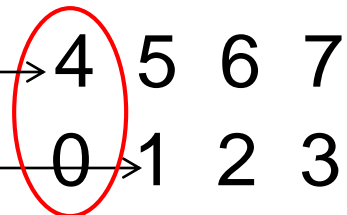


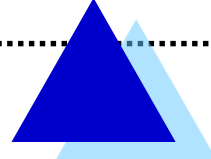


Addressing Memory

- The memory address is formed of a “base” address (stored in a register) and an offset. Since addresses differ by 4, so do the offsets (0, 4, 8, 12, ...). The address of a word matches that of one of its bytes

word address → 4 5 6 7
byte addresses → 0 1 2 3



- Thus the memory address is formed by summing the constant portion of the instruction and the contents of the base register (the second register in the instruction)
- 

Addressing Memory

lw \$t0, 4(\$s3) #what is loaded into \$t0?

sw \$t1, 8(\$s3) # \$t1 is stored where?

If register \$s3 holds 8

\$t1 →
← \$t0

Memory

Data

... 0 1 1 0
... 0 1 0 1
... 1 1 0 0
... 0 0 0 1
... 0 0 1 0
... 1 0 0 0
... 0 1 0 0

24

20

16

12

8

4

0

Word address

Addressing arrays

- Assume the C program contains the instruction
- $a = b + A[4]$
↓ ↓ ↓
\$s1 \$s2 \$t0

First we load $A[4]$ in a temporary register

lw \$t0, 16(\$s3)

Then we add the contents of register \$s2

add \$s1, \$s2, \$t0

A[4]	← \$s3+16
A[3]	← \$s3+12
A[2]	← \$s3+8
A[1]	← \$s3+4
A[0]	← \$s3

Addressing arrays

- Now assume the C program contains the instruction

- $A[7] = a + A[4]$



- `$s1` `$t0`

First we load `A[4]` in a temporary register as before

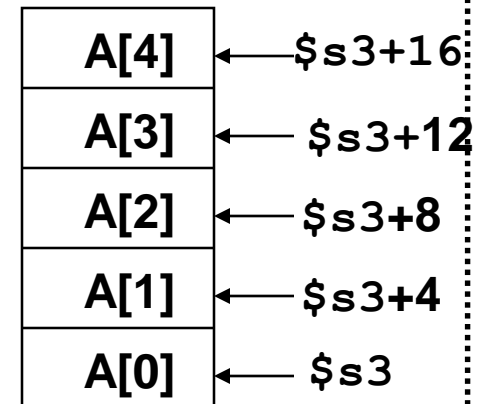
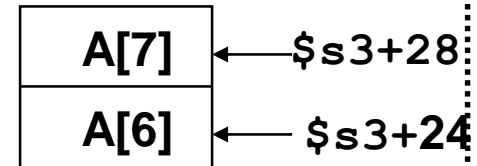
`lw $t0, 16($s3)`

Then we add the contents of registers

`add $t0, $t0, $s1`

Finally, we store back in memory

`sw $t0, 28($s3)`



Addressing arrays

- What if the array location is *variable*? Then we need to compute the address first before loading

- $g = A[i]$



- \$s1, \$s4, and \$s3 holds the base address

First we compute $4i$ doing two additions

$$i + i = 2i; \quad 2i + 2i = 4i$$

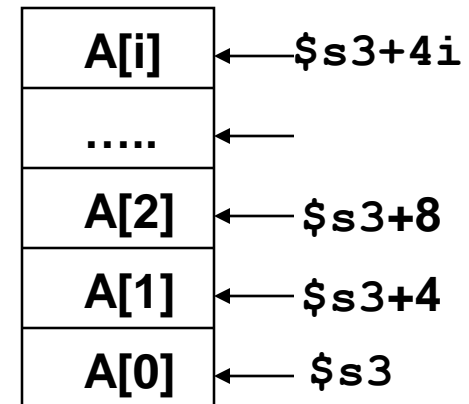
Thus

```
add $t1, $s4, $s4 # $t1 = 2i
```

```
add $t1, $t1, $t1
```

Then we compute the address

```
add $t1, $t1, $s3    lw $s1, 0($t1)
```



Number Representation

- Binary numbers (base 2) - **integers**

0000 → 0001 → 0010 → 0011 → 0100 → 0101 → 0110 → 0111
→ 1000 → 1001 → ...

in decimal from 0 to $2^n - 1$ for **n** bits

MIPS represents numbers as 32-bit constants.

0_{ten} is in MIPS

0000 0000 0000 0000 0000 0000 0000 0000_{two}



Bit 31 (most significant bit) Bit 0 (least significant bit)

Thus the largest (*unsigned*) number is

1111 1111 1111 1111 1111 1111 1111 1111_{two} or

4,294,967,295_{ten} = $2^{32} - 1$

Number Representation

- In any base, a number is represented by a sum. If d is the i^{th} digit, then its value is
 - $d \times \text{Base}^i$ if $i = 2$, as in $300 = 3 \times 10^2$ or $516 = 5 \times 10^2 + 1 \times 10^1 + 6 \times 10^0$
- In MIPS the base is 2
 - Computers use the **two's complement** to represent the *X signed* number as
$$(x_{31} \times -2^{31}) + (x_{30} \times 2^{30}) + \dots (x_1 \times 2^1) + (x_0 \times 2^0)$$
where x_{31} is the most significant bit of X and x_0 is the least significant one.

Number Representation

- Bits are just bits (have no inherent meaning) conventions define the relationships between bits and numbers

- But numbers can also be signed: if the **sign bit** is 0 the number is positive, if it is 1 the number is *negative*.

- Thus the signed number

1111 1111 1111 1111 1111 1111 1111 1111_{two} is
-1_{ten}

- With 32 bits the range of integers becomes

- -2³¹ (-2,147,483,648)

is 1000 0000 0000 0000 0000 0000 0000 0000_{two}

to 2³¹-1 (2,147,483,647)

MIPS Representations

- 32-bit signed numbers (2's complement):

0000 0000 0000 0000 0000 0000 0000 0011_{two} = 3_{ten}

1111 1111 1111 1111 1111 1111 1111 1101_{two} = -3_{ten}

0000 0000 0000 0000 0000 0000 0000 0000_{two} = 3_{ten} - 3_{ten} = 0

- To get the two's complement all 0s become 1s, and all 1s become 0s, then a 1 is added
- Converting n-bit numbers into numbers with more than n bits:
 - MIPS 16-bit immediate gets converted to 32 bits for arithmetic
 - copy the most significant bit (the sign bit) into the other bits

0010 -> 0000 0010

1010 -> 1111 1010

MIPS Representations

- Exercise:

what is the decimal's equivalent of the two's complement of

1111 1111 1111 1111 1111 1110 0000 1100_{two}

- We negate first

0000 0000 0000 0000 0000 0001 1111 0011_{two}
1

0000 0000 0000 0000 0000 0001 1111 0100_{two}

- Then we apply the formula

$$(\mathbf{x}_{31} \times -2^{31}) + (\mathbf{x}_{30} \times 2^{30}) + \dots + (\mathbf{x}_1 \times 2^1) + (\mathbf{x}_0 \times 2^0) =$$

$$1 \times 2^8 + 1 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^2 =$$

$$= 256 + 128 + 64 + 32 + 16 + 4 = 500$$

Hex number representation


Hexadecimal	Binary	Hexadecimal	Binary	Hexadecimal	Binary	Hexadecimal	Binary
0 _{hex}	0000 _{two}	4 _{hex}	0100 _{two}	8 _{hex}	1000 _{two}	c _{hex}	1100 _{two}
1 _{hex}	0001 _{two}	5 _{hex}	0101 _{two}	9 _{hex}	1001 _{two}	d _{hex}	1101 _{two}
2 _{hex}	0010 _{two}	6 _{hex}	0110 _{two}	a _{hex}	1010 _{two}	e _{hex}	1110 _{two}
3 _{hex}	0011 _{two}	7 _{hex}	0111 _{two}	b _{hex}	1011 _{two}	f _{hex}	1111 _{two}

- Allows a more compact representation of numbers – each group of 4 bits corresponds to 1 hex number

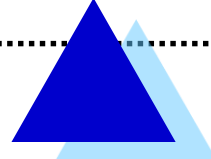
f f a 9 7 c 4 0_{hex}

- Corresponds to the binary string

1111 1111 1010 1001 0111 1100 0100 0000_{two}



Representing Instructions in Machine Language

- Instructions, like registers and words of data, are also *32 bits long*
 - They are formed by placing binary number “fields” side-by-side to form a 32 bit machine language instruction
 - The meaning of these bits (or the “format” of the instruction) depends on its type.
 - *Instruction Format differs* for R-type instructions (add, sub) vs. I-type instructions (lw, sw) – data access type
- 

Machine Language Instructions

- The first field is the **op field** (6 bits), which tells the control block in the processor what *type* of instructions this is (but not always what instruction it is)
- For R-type the op field is **0s**, for **lw** it is binary equivalent of 35, for **sw** it is the binary equivalent of 43

op					
----	--	--	--	--	--

R-type

000000					
--------	--	--	--	--	--

op			
----	--	--	--

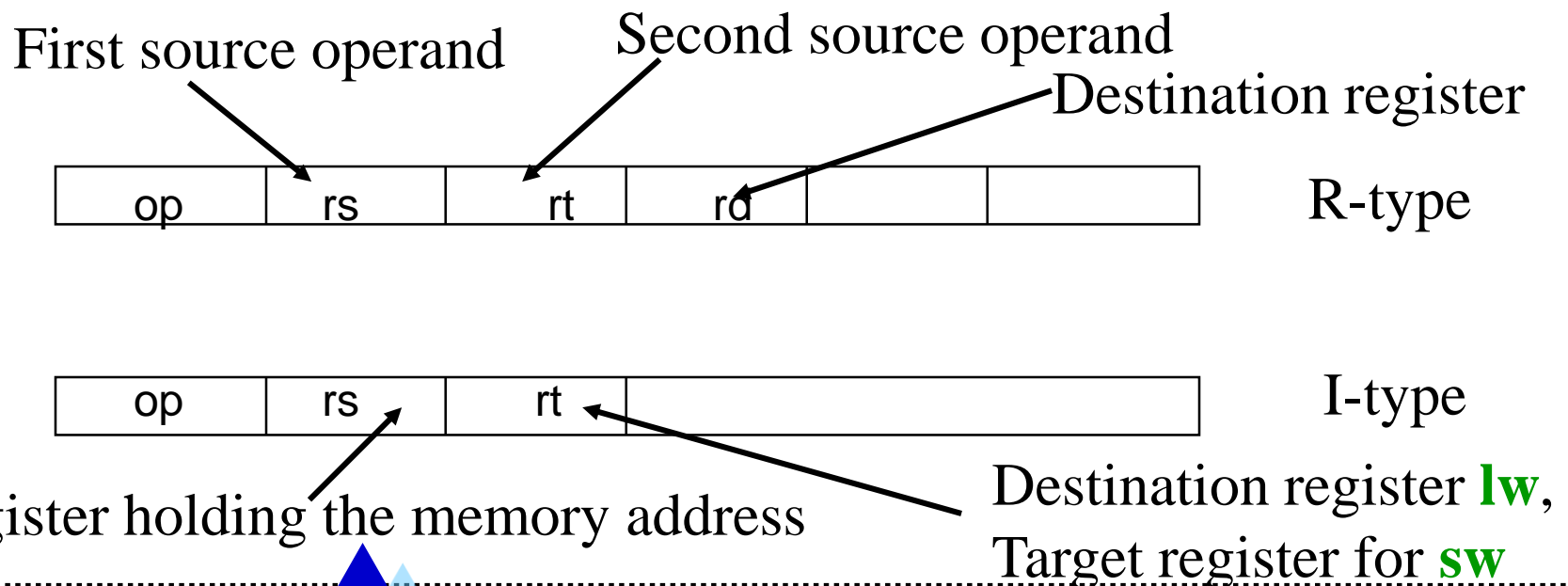
I-type (**lw**)

100011			
--------	--	--	--

35 ↗

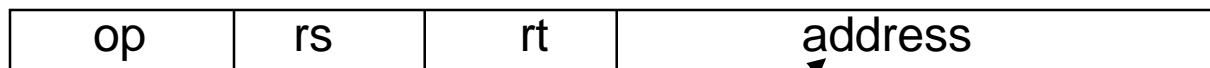
Machine Language Instructions

- The second field is the **rs field** (5 bits long);
- The third field (also 5 bits) is “rt” – it is the second source operand for R-type and the destination register for I-type



Machine Language Instructions

- The last field in I-type operations is the 16 bit-long “address” *offset* field which is a constant.
- that means the offset has a max value of $\pm 2^{15}$ (32,768 bytes) or $\pm 2^{13}$ words ($\pm 8,192$ row locations)



I-type

16-bit offset constant



Machine Language Instructions

- **Example**

sw \$t0, 24(\$s2)

We know that \$t0 is 8 and \$s2 is 18, so

op	rs	rt	16 bit number
----	----	----	---------------



43	18	8	24
----	----	---	----



101011	10010	01000	00000000000011000
--------	-------	-------	-------------------

Machine code





Machine Language Instructions

- The fourth field for R-type instructions is the **rd field** (5 bits), which is the destination operand – the name of the register where results are stored.
- The fifth field is the “**shift amount**” (5 bits) - unused (zeros);
- The last field of 6 bits for R-type operations is the “**function**” **field** – tells the processor what kind of R-type operation to perform (it is 32 for **add**, 34 for **sub**, etc.)

op	rs	rt	rd	shamt	funct
----	----	----	----	-------	-------

R-type



Machine Language Instructions

- What is the machine code for:

sub \$t0, \$s1, \$s2 #t0=s1-s2 ?

- First we need the register addresses

\$t0, ... \$t7 and \$s0 ... \$s7 in MIPS have addresses
8 ... 15 16 ... 23

- So $rs = \$s1 = 17_{10} = 10001$; $rt = \$s2 = 18_{10} = 10010$
 $rd = \$t0 = 8_{10} = 01000$; $funct = 34_{10} = 100010$

op	rs	rt	rd	shamt	funct
000000	10001	10010	01000	00000	100010

Immediate Instructions

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

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

- Solutions?
- put “typical constants” in memory and load them
- To avoid frequent loads, architects created a hard-wired register (**\$at** – address 1) for *constants*

Immediate Instructions

- MIPS instruction for adding immediate values. For example, if we want to increment or decrement the stack pointer:

addi \$sp, \$sp, 4 # \$sp = \$sp + 4

addi \$sp, \$sp, -4 # \$sp = \$sp - 4

- This is another version of add in which one operand is a constant where the constant is kept inside the instruction itself. How do we make it work?

Immediate operands

- MIPS **immediate** instructions:

addi \$sp, \$sp, 8 # \$sp = \$sp + 8

- Machine format:

op	rs	rt	16 bit immediate
----	----	----	------------------

 / format

8	29	29	8
---	----	----	---



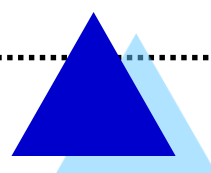
001000	11101	11101	0000 0000 0000 1000
--------	-------	-------	---------------------

- Immediate constants are signed. **Limits** immediate values to the range $+2^{15}-1$ to -2^{15}



MIPS Instructions, so far

Category	Instr	Op Code	Example	Meaning
Arithmetic (R format)	add	0	add \$s1, \$s2, \$s3	$\$s1 = \$s2 + \$s3$
	subtract	0	sub \$s1, \$s2, \$s3	$\$s1 = \$s2 - \$s3$
Data transfer (I format)	load word	35	lw \$s1, 100(\$s2)	$\$s1 = \text{Memory}(\$s2+100)$
	store word	43	sw \$s1, 100(\$s2)	$\text{Memory}(\$s2+100) = \$s1$
	addi	8	addi \$sp, \$sp, 8	$\$sp = \$sp + 8$





Machine Language Instructions

- What is the machine code for
 $A[300] = h - A[300]$ in C. Assume h is stored in $\$s2$ and
base address in $\$s3$. Remember $\$s3$ is 19,
- First the corresponding assembly code is

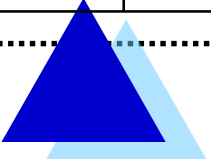
```
lw $t0, 1200($s3) #t0 gets A[300]
sub $t0, $s2, $t0 #t0 gets h-A[300]
sw $t0, 1200($s3) #store t0 back in memory
```

Intermediate decimal notation

35	19	8	1200
----	----	---	------

0	18	8	8	0	34
---	----	---	---	---	----

43	19	8	1200
----	----	---	------





Machine Language Instructions

- What is the machine code for
 $A[300] = h - A[300]$ in C?

100011	10011	01000	0000 0100 1011 0000
--------	-------	-------	---------------------

000000	10010	01000	01000	00000	100010
--------	-------	-------	-------	-------	--------

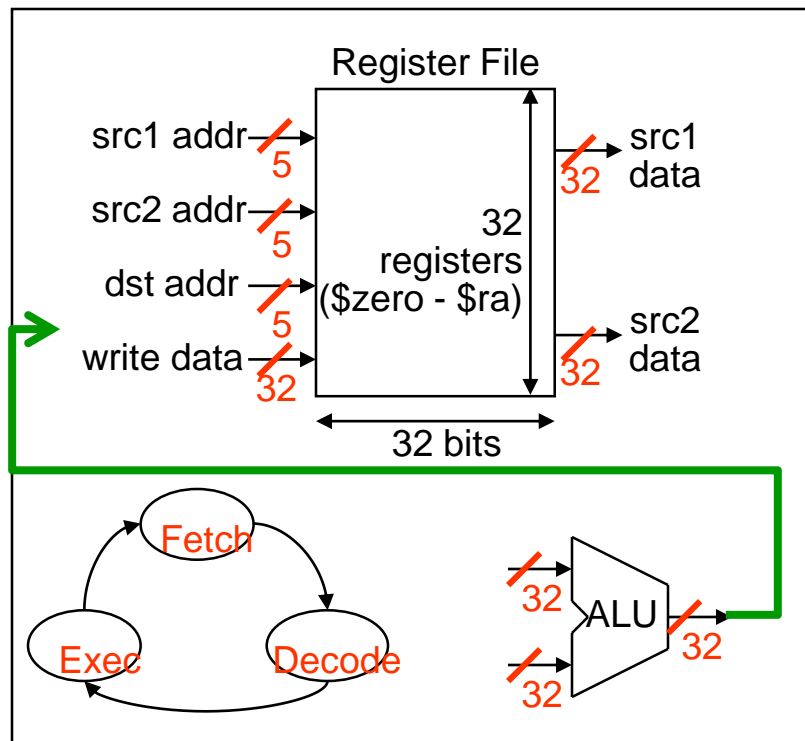
101011	10011	01000	0000 0100 1011 0000
--------	-------	-------	---------------------

- Instructions are numbers – can be stored in memory like data – “store program” concept
- 

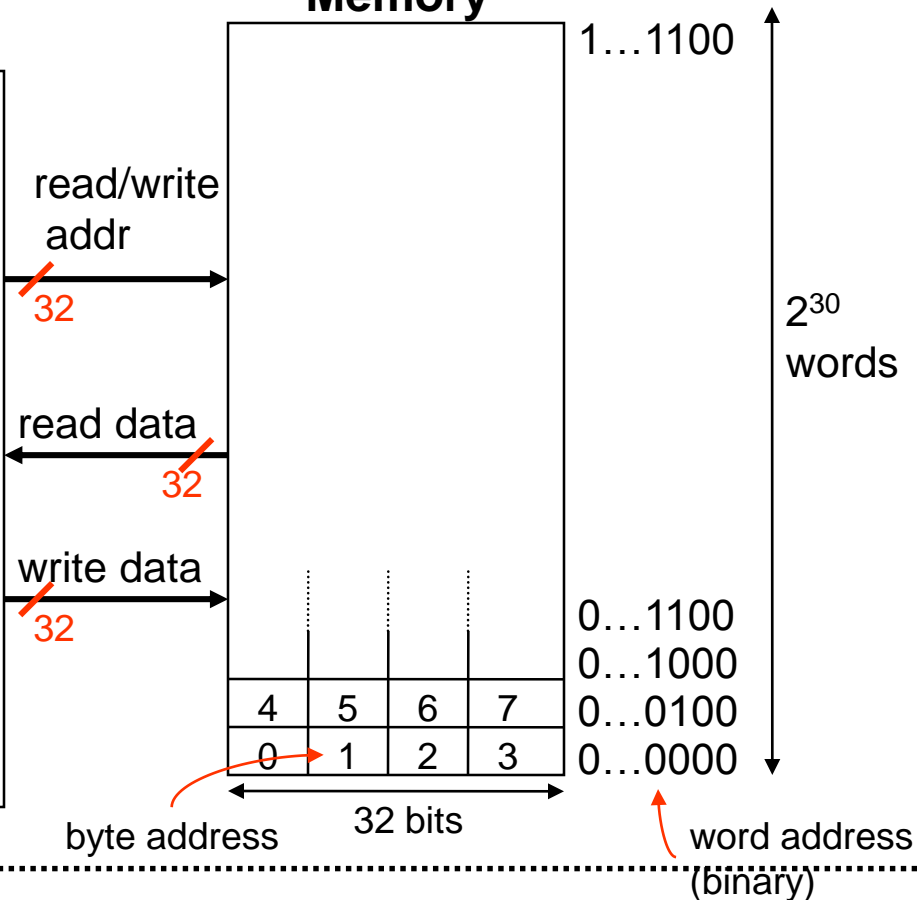
Instructions, so far

- Arithmetic instructions – to/from the register file
- Load/store word instructions – from/to memory
- Instructions are stored like data, simplifies memory hardware – *stored program concept*.

Processor



Memory



Logical operations

- Operate on field of bits and on individual bits
- Needed, for example when examining characters in a word, but also in multiplying numbers

Logical Op	C operator	Java	MIPS instruction
Shift left	<<	<<	sll (used in multiplications)
Shift right	>>	>>> (unsigned)	srl (used in divisions)
AND (bit-by-bit)	&	&	and, andi
OR			or, ori
NOR	~	~	nor

Shifts

- Let us consider the decimal number
- 123_{10}
- Adding two 00 at the end shifts the number to the left
- 12300_{10}
- It is equivalent to multiplying the number with 10^2
- So for every shift left by 1 it is like we multiply by 10.
- In binary, shifting left by i bits adds 0s at the end.
- It is like multiplying the number with 2^i
- $0010 = 2_{10}$ shifted left by 2 is like multiplying by $2^2 = 4$
- 1000 after shift = 8_{10} (2×4)

Logical Operations

- Logical shift operations move all the bits of a word to the *left* or to the *right* a specified number of bits. The emptied bits are filled with **0**s. It is an R-type instruction

sll \$t2, \$t1, 4 # \$t1 << 4

- places in \$t2 the contents of register \$t1 shifted left by 4 bits. **Shamt** = shift amount

sll	0	0	\$t1	\$t2	shamt	0	R-type
srl	0	0	\$t1	\$t2	shamt	2	R-type

Ex. \$t1 = 0000 0001... 0000 1111
 \$t2 = 0001... 1111 **0000** after **sll 4**
 \$t2 = **00**00... 0000 0011 after **srl 2**

Logical Operations

- To apply a bit pattern – to force 0s on a bit field where there are 0s in the bit pattern

and \$t0, \$t1, \$t2

places in \$t0 the logical AND of registers \$t1 and \$t2.

- Logical **and** puts a 1 in the field where **both** \$t1 and \$t2 bits were 1s at that location

0	\$t1	\$t2	\$t0	0	36
---	------	------	------	---	----

R-type

- Example $\$t1 = 0000\dots 0000\ 1101$
 $\$t2 = 1000\dots 0000\ 1001$
 $\$t1 \& \$t2 = 0000\dots 0000\ 1001$

Logical Operations

- The dual of AND is OR

or \$t0, \$t1, \$t2

places in \$t0 the logical **or** of contents of registers \$t1 and \$t2.

- Logical **or** puts a 1 in the field where **either** \$t1 and \$t2 bits at that location were 1s

0	\$t1	\$t2	\$t0	0	37	R-type
---	------	------	------	---	----	--------

- example \$t1 = 0000... 0000 1101
\$t2 = 1000... 0000 1001
\$t1 | \$t2 = 1000... 0000 1101

Logical Operations

- **NOT (A)** places a 1 in result if A bit is 0 and places 0 in result if A bit is 1. **NOR** is a MIPS instruction
- **A NOR 0 = NOT (A OR 0) = NOR (A)**
nor \$t0, \$t1, \$t2
places in \$t0 the logical nor of registers \$t1 and \$t2.
- Logical **nor** (*not or*) puts a 0 in the field where **either** \$t1 and \$t2 bits were 1s

0	\$t1	\$t2	\$t0	0	39
---	------	------	------	---	----

R-type

- example $\$t1 = 0000\dots 0000 \ 1101$
 $\$t2 = 1000\dots 0000 \ 1001$
 $\sim(\$t1 \mid \$t2) = 0111\dots 1111 \ 0010$

Logical Instructions

- There are also immediate versions of logical operations
`andi $s1, $s2, 1000`
`ori $s1, $s2, 1000`
- Places in register \$s1 the **and** or the **or** of register \$s2 with a 16-bit constant (the *least significant bits* are & or |). This works like a “mask” to cover or uncover fields.

andi

12	\$s2	\$s1	constant
----	------	------	----------

I-type

ori

13	\$s2	\$s1	constant
----	------	------	----------

I-type

- ori** is also used in conjunction with **lui** to load a 32-bit constant in a register
- While **and** and **or** are R-type, **andi** and **ori** are I-type

Control Flow Instructions



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

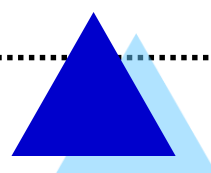
- Why do we need decision making instructions?

```
if (i==j) h = i + j;
```

- MIPS **conditional branch** instructions:

```
bne $s0, $s1, Lab1    #go to Lab11 if  
value stored in $s0≠$s1
```

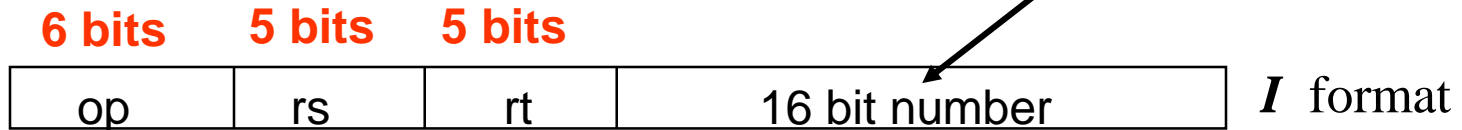
```
beq $s0, $s1, Lab2    #go to Label2 if  
value stored in $s0 equals that stored in  
$s1
```



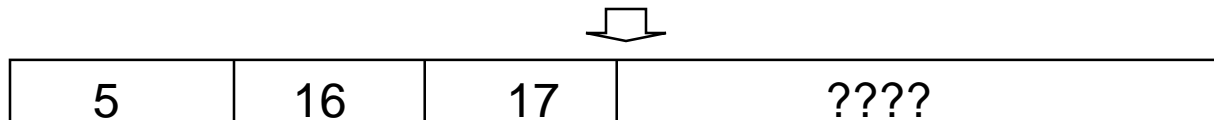
Conditional branch instructions

- Recall that saved register $\$s0$ corresponded to address 16 in register file and $\$s1$ to address 17
- Machine Formats:

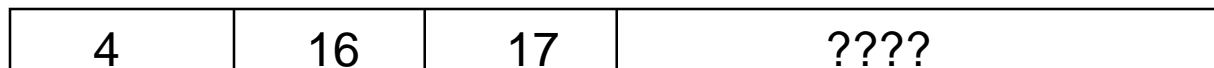
Specifies the branch destination



bne

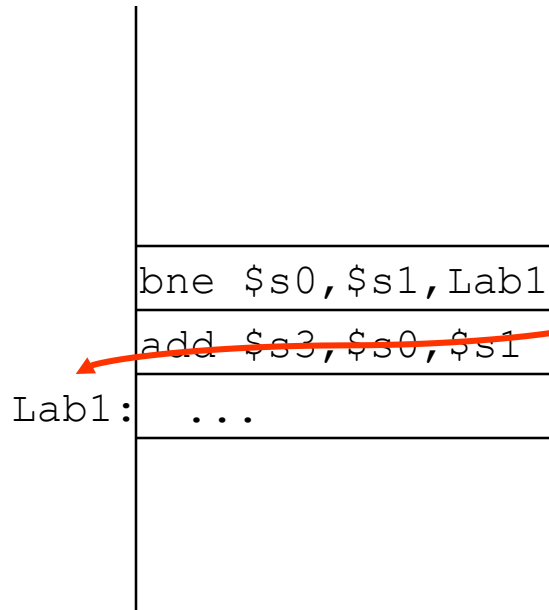


beq



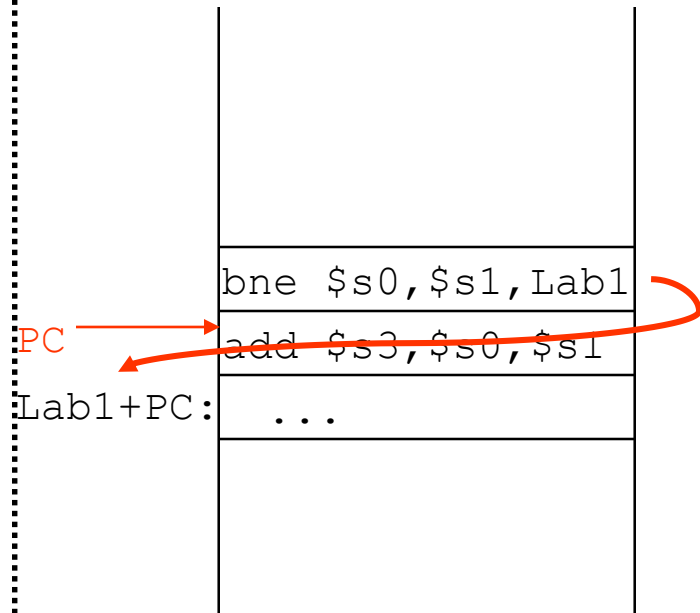
- How is the branch destination address specified?

Specifying Branch Destinations



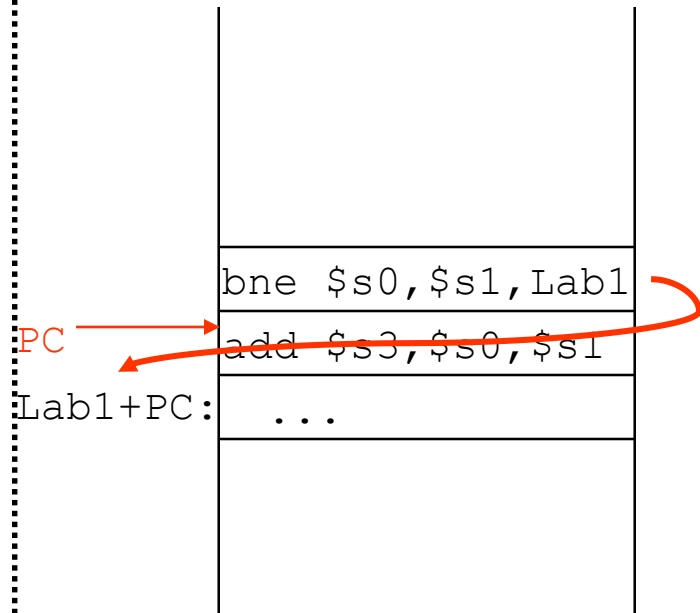
- Could specify the memory address
- but that would limit the program size to 2^{16} instructions
- How can conditional branches operate inside *large* programs?
- We know that conditional branches typically go to addresses close to where the test was done.

Specifying Branch Destinations



- MIPS architecture uses **program counter** *PC-relative addressing*
- PC gets updated ($PC+4$) during the **fetch** cycle so that it holds the address of the *next* instruction
- This allows jumps of -2^{15} to $2^{15} - 1$ from the *next* instruction after conditional branch

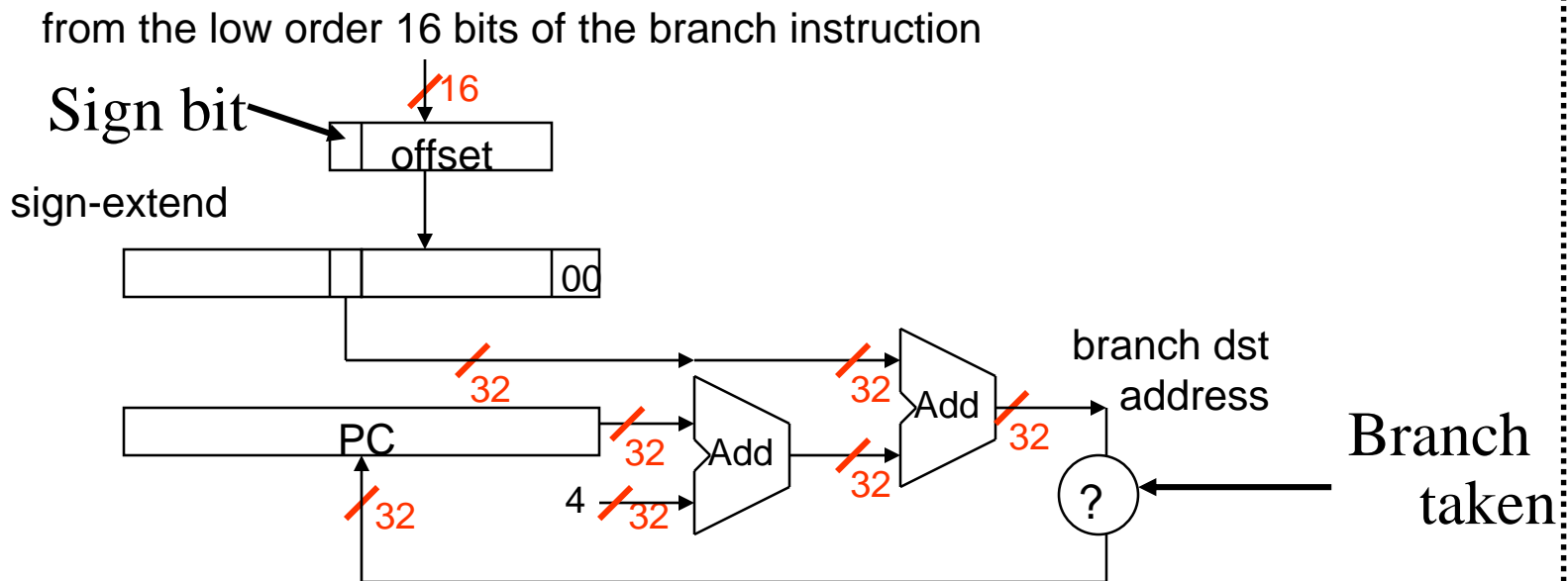
Specifying Branch Destinations



- We can optimize since address is multiple of 4 (a word address) –
- Since 4_{10} in binary is 100 , all word address have **00** as last two bits – 8_{10} is 1000 , 12_{10} is 1100 , etc.
- Optimization - right shift the offset by **2 bits** (divided by 4), and store the value
- Essentially, it can cover -2^{17} to $+2^{17}-1$ offset

Computing Branch Destinations

- The contents of PC+4 are added to the low order 16 bits of the branch instruction which is converted into a 32 bit value: concatenating two low-order zeros to create an 18 bit number
 - Then sign-extending those 18 bits



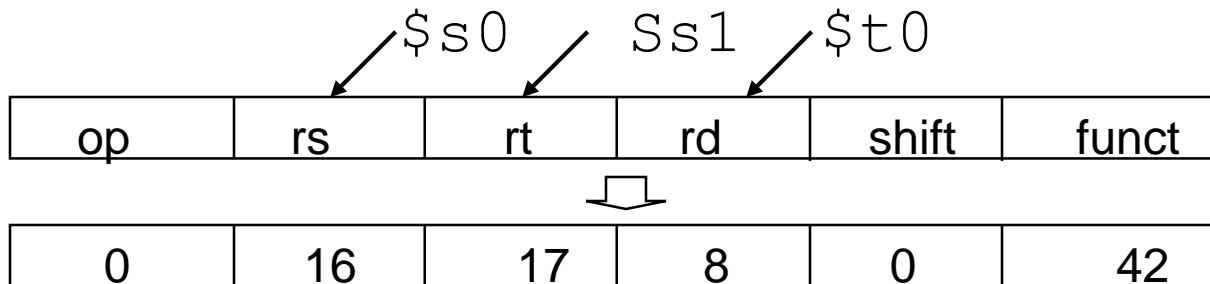
- The result is written into the PC *if* the branch condition is true prior to the next Fetch cycle

More control flow instructions

- We have `beq`, `bne`, but what about branch-if-less-than?
- New R-type instruction called “set on less than”

slt `$t0, $s0, $s1`

- if value stored in `$s0` is *less than* the value stored in `$s1`
then set contents of `$t0` to **1** else set `$t0` = 0



R-type
instruction

- New I-type instruction called “set on less than immediate”

slti `$t0, $s0, 20`

- if value stored in `$s0` is *less than* 20 then set contents of `$t0` to **1**
else set `$t0` = 0



I-type instruction



Comparing signed and unsigned integers

- Set-on-less-than **slt** and set-on-less-than-immediate **slti** compare two *signed* integers (MSB is the sign bit).
- To tell the control block that it is handling unsigned numbers, there are two other instructions
- “Set-on-less-than-unsigned **sltu** and
- “Set-on-less-than-immediate-unsigned” **sltiu**

0	16	17	8	0	43
---	----	----	---	---	----

11	rs	rd	constant
----	----	----	----------

I-type instruction





Unconditional branch instructions

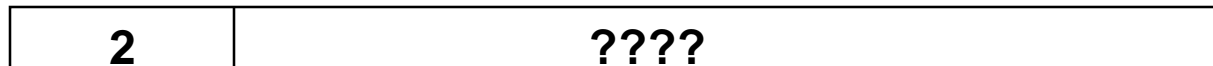
- There are two types:

jump instruction:

j exit #go to the statement with label
"exit"



- Machine Format:

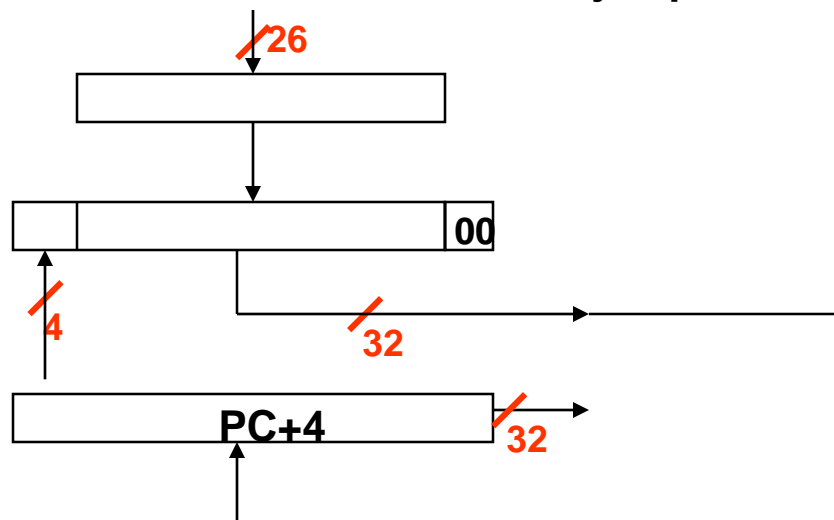


- How is the jump destination address specified?
- 

Computing Jump Destination

- Is an absolute address formed by concatenating the upper 4 bits of the current PC (now PC+4) to the 26-bit address and concatenating 00 as the 2 low-order bits

from the low order 26 bits of the jump instruction



- Creates a 32 bit instruction address that is placed into the PC prior to the next Fetch cycle (no testing) – Forced!

Jump instructions - continued

- Instruction:
jump register instruction
- **jr** \$t1 #go to the address *stored* in register \$t1
- Machine Format:

op	\$t1	0	0	0	funct
----	------	---	---	---	-------

R format

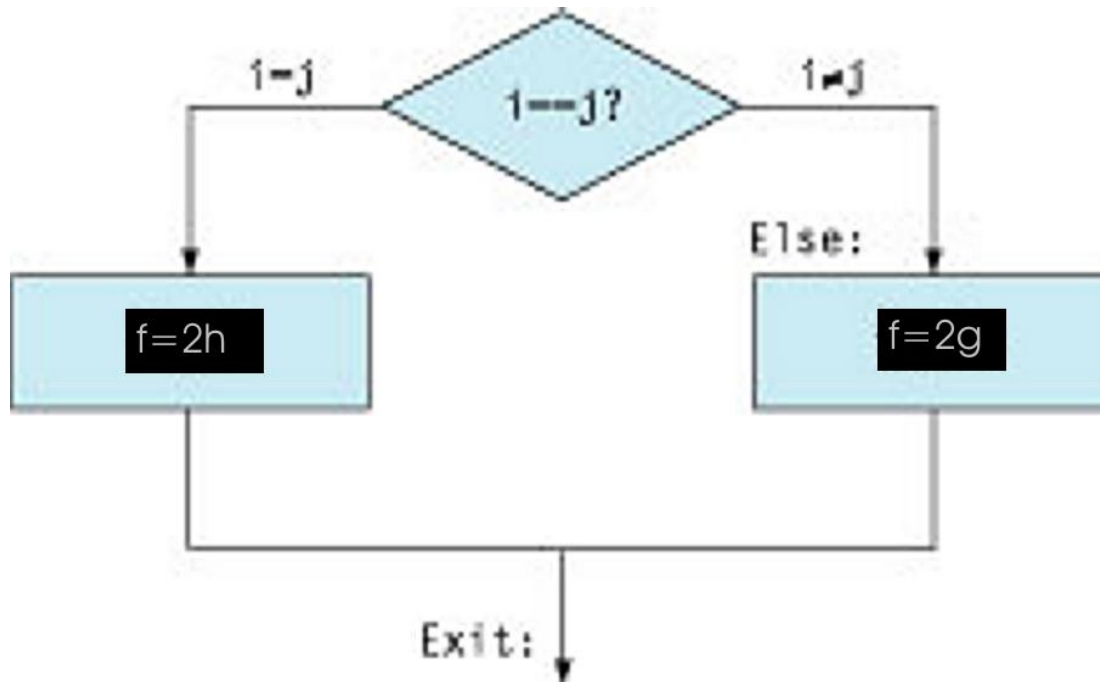


0	9	0	0	0	8
---	---	---	---	---	---

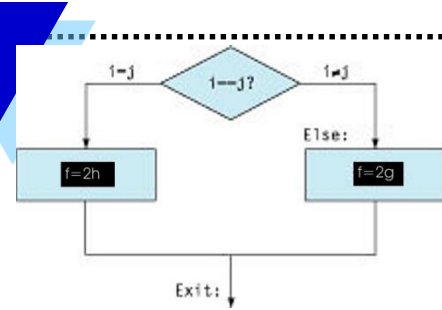
- This is “*register addressing*”, where the operand is a register, vs. PC-relative addressing used in branch instructions

Combining Branches and Jumps

- In a basic program block branch instructions are at the end. Example – the high-level `if` statement:
- `if (i==j) f=2*h else f=2*g;`



Combining Branches and Jumps



- In MIPS it is more efficient to test for **bne**.
- assume that *f* is stored in *\$s0*, *g* in *\$s1*, *h* in *\$s2*, *i* in *\$s3*, and *j* in *\$s4*
- The above statement is implemented in assembly as:
bne *\$s3*, *\$s4*, ELSE #if number in *\$s3* is
not equal to number in *\$s4* go to ELSE
add *\$s0*, *\$s2*, *\$s2* # *f*=2**h*
j EXIT
ELSE: add *\$s0*, *\$s1*, *\$s1* #*f*=2**g*
EXIT:

Loops - Combining Branches and Jumps

- Another example – the high-level while loop:

```
while (A[i]==k) i=2*i;
```

We assume that base address of A is in \$s5, i in \$s3, k in \$s4

- The above statement is implemented in assembly as:

- Loop: **sll** \$t1, \$s3, 2 # \$t1=4i

instead of 2 additions of add \$t1, \$s3, \$s3

#and then add \$t1, \$t1, \$t1

add \$t1, \$t1, \$s5 # set address for A[i]

lw \$t0, 0(\$t1) # \$t0=A[i]

bne \$t0, \$s4, EXIT #if A[i] not eq to k

addi \$s3, \$s3, 1 # i=i+1

j Loop # return to Loop

EXIT:

Exercise

- Re-write the assembly code for the high-level while loop:

`while (save[i]==k) i=i+j;` Such that it has at most one jump or one branch each time through the loop. How many instructions are executed in both cases if 10 loops are done. We assume that base address of `save[i]` is stored in `$s6`, `i` in `$s3`, `j` in `$s4`, `k` in `$s5`

- Before optimization:

```
Loop: add $t1, $s3,$s3 # $t1=2i
      add $t1, $t1, $t1 # $t1=4i
      add $t1, $t1, $s6 # set address for save[i]
      lw  $t0, 0($t1)   # $t0=save[i]
      bne $t0, $s5, EXIT #if save[i] not equal to k
      add $s3, $s3, $s4 # $s3=i+j or i=i+j
      j   Loop # return to Loop
```

EXIT:

- Number of instructions = $10 \times 7 + 5 = 75$

Exercise - continued

- After optimization:

```
sll $t2, $s4, 2 # $t2 = 4j
```

```
sll $t1, $s3, 2 # $t1 = 4i
```

```
add $t1, $t1, $s6 # $t1 has base address of save[i]
```

```
lw $t0, 0($t1) # $t0 = save[i]
```

```
bne $t0, $s5, EXIT # if save[i] != k
```

```
Loop: add $t1, $t1, $t2 # $t1 has address of save[i+m*j]
```

```
lw $t0, 0($t1) # $t0 = save[i]
```

```
beq $t0, $s5, Loop # loop if save[i] = k
```

EXIT:

- Number of instructions = $10 \cdot 3 + 5 = 35$

Other Branch Instructions

- Can use `slt`, `beq`, `bne`, and the fixed value of 0 in register `$zero` to **create** all relative conditions

less than **blt** `$s1, $s2, Label`

less than or equal to **ble** `$s1, $s2, Label`

greater than **bgt** `$s1, $s2, Label`

great than or equal to **bge** `$s1, $s2, Label`

- These are *pseudo instructions* recognized (and expanded) by the assembler
- The assembler needs a reserved register (**\$at**)
there are policies of use conventions for registers

Pseudo Instructions

- **Example** - produce a minimal sequence of actual MIPS instructions to accomplish what the pseudo instruction means to do:

ble \$t5, \$t3, Label #if number in \$t5 is \leq number in \$t3, branch to Label

- The equivalent MIPS instructions are:

```
slt $at, $t3, $t5 #if $t3 < $t5, $at=1
beq $at, $zero, Label
```

- Another pseudo instruction

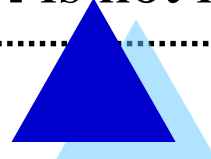
bgt \$t5, \$t3, Label #if number in \$t5 is $>$ number in \$t3, branch to Label

- The equivalent MIPS instructions are:

```
slt $at, $t3, $t5 #if $t3 < $t5, $at=1
bne $at, $zero, Label
```

Case statement



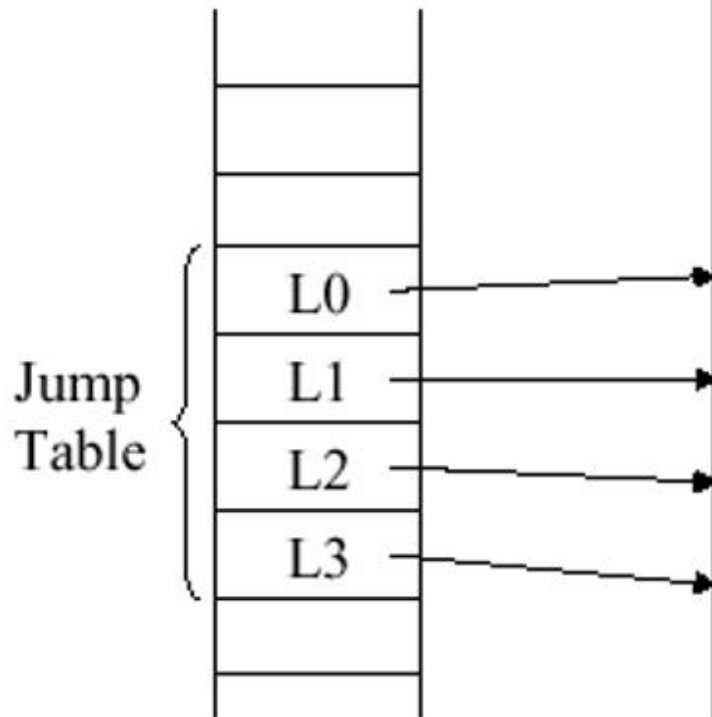
- Most higher level languages have case or switch statements allowing the code to select one of many alternatives depending on the value of a single variable – Which is the *case*?
 - This is done with a **jump address table**, an array of words containing addresses corresponding to the labels in the code.
 - ```
switch (k) {
```
  - ```
    case 0: f=i-j; break;
```
 - ```
 case 1: f=g-h; break;
```
  - ```
    case 2: f=g+h; break'
```
 - ```
 case 3: f=i+j; break;
```
  - ```
}
```
 - We assume that *f* is in *\$s0*, *g* in *\$s1*, *h* in *\$s2*, *i* in *\$s3*, *j* in *\$s4*, *k* in *\$s5*, *\$t2* has 3 in it and the jump table base address is in *\$t4*. If *k* is not in the range 0-3, the switch should exit.
- 

- The corresponding assembly code is

```
slt $t3, $s5, $zero #test if k<0 (k was stored in
$s5)
bne $t3, $zero, EXIT #if #t3 is 1 exit since k<0
slt $t3, $s5, $t2 #test if k>3 (3 was stored in $t2)
beq $t3, $zero, EXIT #if k>3, $t3=0, exit
sll $t1, $s5, 2 # $t1=4*k
add $t1, $t1, $t4 # $t1 now holds address of jump
table[k]
lw $t0, 0($t1) # $t0 now has address of label Li
jr $t0 #go to appropriate label address
L0: sub $s0, $s3, $s4 #f=i-j if k=0
j EXIT
L1: sub $s0, $s1, $s2 #f=g-h if k=1
j EXIT
L2: add $s0, $s1, $s2 #f=g+h if k=2
j EXIT
L3: add $s0, $s3, $s4 #f=i+j if k=3
EXIT:
```

Jump address table

- Jump register instruction
 - jr register
 - unconditional branch to address contained in register



```
# f: $s0; g: $s1; h: $s2; i: $s3; j: $s4; k:$s5
# $t2 = 4; $t4 = base address of JT

slt    $t3, $s5, $zero    # test k < 0
bne    $t3, $zero, Exit   # if so, exit
slt    $t3, $s5, $t2      # test k < 4
beq    $t3, $zero, Exit   # if so, exit
add    $t1, $s5, $5       # $t1 = 2*k
add    $t1, $t1, $t1      # $t1 = 4*k
add    $t1, $t1, $t4      # $t1 = &JT[k]
lw     $t0, 0($t1)        # $t0 = JT[k]
jr     $t0                # jump register

L0: add $s0, $s3, $s4      # k == 0
j      Exit               # break
L1: add $s0, $1, $s2       # k == 1
j      Exit               # break
L2: sub $s0, $s1, $s2      # k == 2
j      Exit               # break
L3: sub $s0, $s3, $s4      # k == 3
Exit:
```

Procedures (subroutines)



- A procedure that does not call another procedure is called a “leaf”

```
void main() {  
    int    f;  
    f = leaf_example(1, 2, 3, 4);  
    f ++;  
}
```

CALLER

```
int leaf_example (int g, int h, int i, int j) {  
    int    f;  
    f = (g-h) + (i-j);  
    return f;  
}
```

CALLEE



Name Convention for Registers

Registers 0.. 15

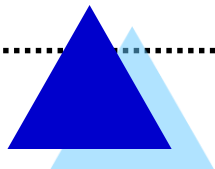


0	\$zero constant 0
1	\$at reserved for assembler
2	\$v0 expression evaluation &
3	\$v1 function results
4	\$a0 argument registers
5	\$a1
6	\$a2
7	\$a3
8	\$t0 temporary: caller saves
...	(callee can clobber)
15	\$t7



Six Steps in Execution of a Procedure

- Main routine (**caller**) places actual parameters in a place where the procedure (**callee**) can access them
\$a0 - **\$a3**: four *argument registers*
- Caller transfers control to the callee
- Callee acquires the storage resources needed and performs task
- Callee places the results in a place where the caller can access it - **\$v0** - **\$v1**: two *return value registers*
- Callee does more resource management and then returns control to the caller
- How does MIPS know where to return in the caller program?



Name Convention for Registers

Registers 16... 31

16 **\$s0** callee saves

... 18 **\$s2**

23 **\$s7**

24 **\$t8** temporary (cont'd)

25 **\$t9**

26 **\$k0** reserved for OS kernel

27 **\$k1**

28 **\$gp** pointer to global area

29 **\$sp** stack pointer

30 **\$fp** frame pointer

31 **\$ra** return address

Flow control for calling a procedure

- MIPS uses another type of jump instruction called “**jump and link**” placed in the caller program

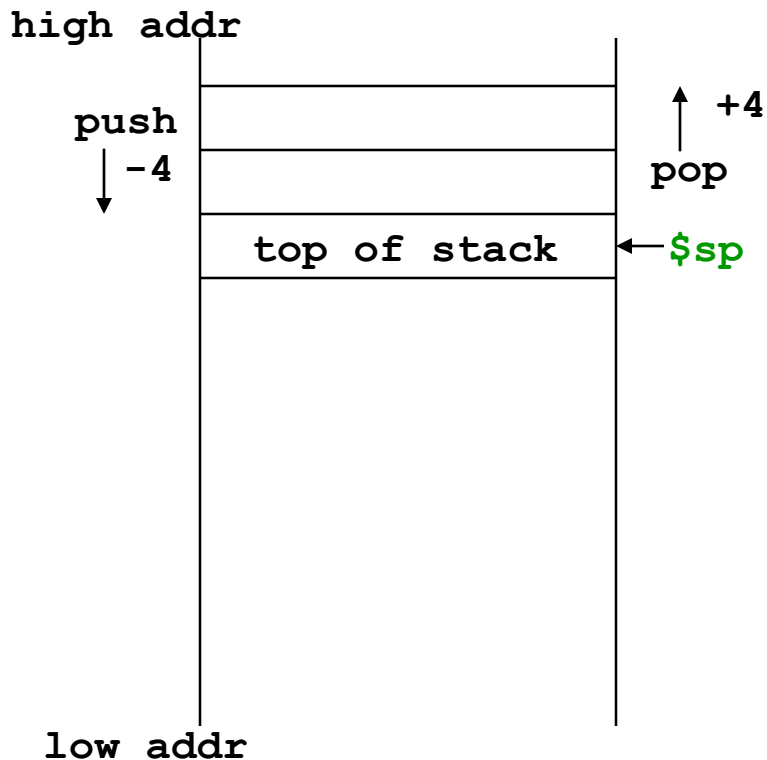
jal ProcedureAddress #jump to callee address



- **j** had opcode 2. **jal** a different opcode
- When **jal** is executed, the current value of the program counter (PC+4) is saved in a dedicated register called “return address register” **\$ra** (for **j** instruction this was not done)
- Once callee is finished it returns control to the caller by using a **jr \$ra** #return

Stack Pointer

- Where does the callee save “saved registers” (\$s . . registers) before starting execution?
- “Spilled” registers are saved in memory using a **stack** – a last-in-first-out queue



- The stack pointer register, **\$sp** is used to address the stack (which “grows” from high address to low address)

- add data onto the stack – **push**

$$\begin{aligned} \mathbf{\$sp} &= \$sp - 4 \\ &\text{data on stack at new } \$sp \end{aligned}$$

- remove data from the stack – **pop**

data from stack at \$sp

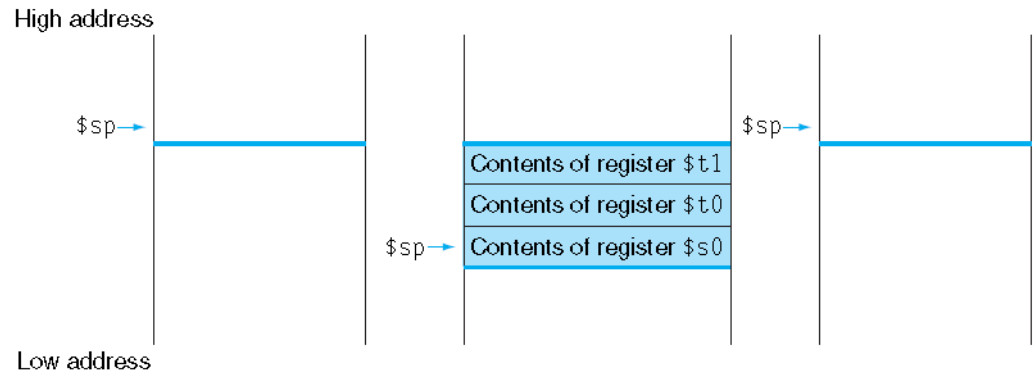
$$\mathbf{\$sp} = \$sp + 4$$

Procedure Example

```
int leaf_example (int g, int h, int i, int j)
{
    int    f;
    f = (g-h) + (i-j);
    return f;
}
```

- Argument registers hold the variables *g* in \$a0, *h* in \$a1, *i* in \$a2 and *j* in \$a3. *f* is stored in \$s0.
- The procedure uses the stack to store values of saved registers in memory, *before* performing its task
- The caller restores the *old* values of the saved registers for use by the main program before returning.

Procedure Example (assembly)



Leaf:

```
subi  $sp, $sp, 12 # $sp=$sp-12
sw    $t1, 8($sp) #save the value of $t1
sw    $t0, 4($sp) #save the value of $t0
sw    $s0, 0($sp) # save the value of $s0
sub    $t0, $a0, $a1 # $t0 now contains g-h
sub    $t1, $a2, $a3 # $t1 now contains i-j
add    $s0, $t0, $t1 # $s0 now contains (g-h)+(i-j)
add    $v0, $s0, $zero # places result in register $v0
lw     $s0, 0($sp) #restores the old value of $s0
lw     $t0, 4($sp) #restores the old value of $t0
lw     $t1, 8($sp) #restores the old value of $t1
addi   $sp, $sp, 12 #reset the value of stack pointer
jr     $ra
```

Normally temporary register values are not saved

Nested Procedure

- When a procedure calls another one, which in turn calls another ..
- This results in conflict in argument registers that hold the variables `$a0`, `$a1`, `$a2` and `$a3`. The *caller* pushes them as well as `$t0` .. `$t9` into the stack.
- There is also conflict in the return address register `$ra`
- The contents of the `$ra` register together with the contents of saved registers need to be saved in stack by the *calee*, and then restored.

- Example $i! = i * (i-1) * \dots * 1$ which in C is

```
int fact (int n)
{
    if (n<1) return(1);
    else return(n*fact(n-1));
}
```

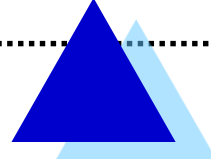
- The variable **n** is the argument passed in **\$a0**. It needs to be saved in stack, together with the contents of the **\$ra**



Nested Procedure Example

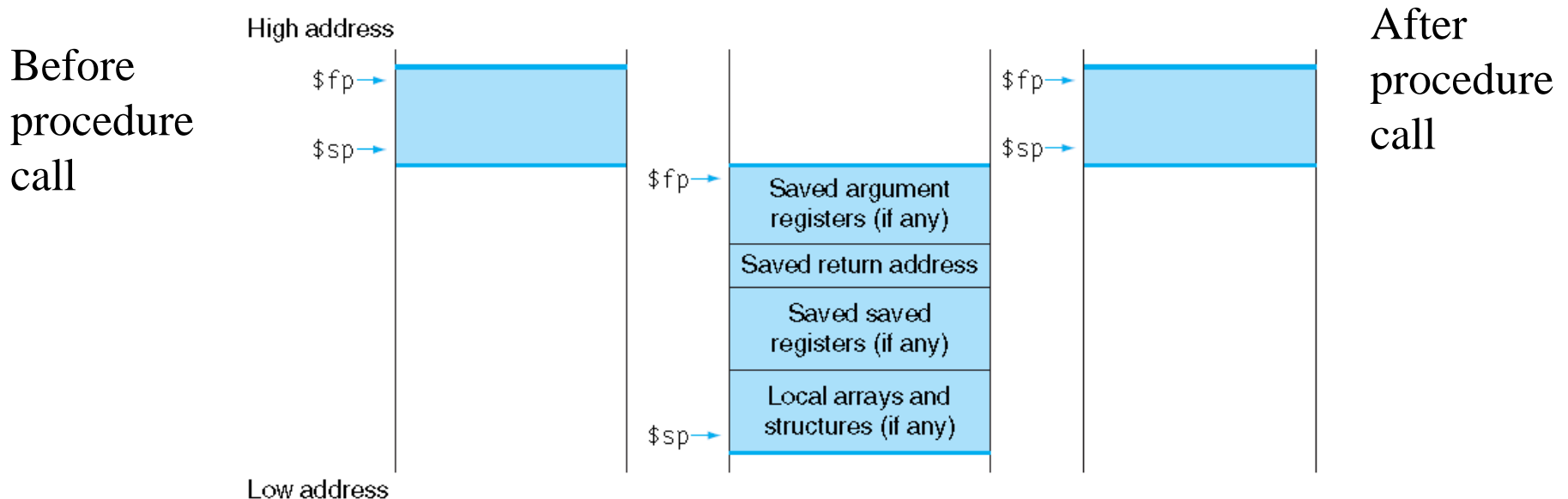
Fact:

```
subi $sp, $sp, 8 # $sp=$sp-8
sw   $ra, 4($sp) #save the value of $ra
sw   $a0, 0($sp) #save value of argument register
slti $t0, $a0, 1 # test if n<1
beq  $t0, $zero, L1 #if n≥1 go to L1
addi $v0, $zero, 1 #return value of 1
addi $sp, $sp, 8 #restore the stack pointer
jr   $ra #return to the instruction after jal ($ra and
        $a0 were not loaded back from stack)
L1: subi $a0, $a0, 1 #n=n-1 (as long as its >1)
    jal  Fact #call fact(n-1), and return at the next
instruction (lw)
    lw   $a0, 0($sp) #restores the old value of $a0
    lw   $ra, 4($sp) #restores the old value of $ra
    addi $sp, $sp, 8 #reset the value of stack pointer
    mul  $v0, $a0, $v0 # n*fact(n-1)
    jr   $ra #return to the caller
```

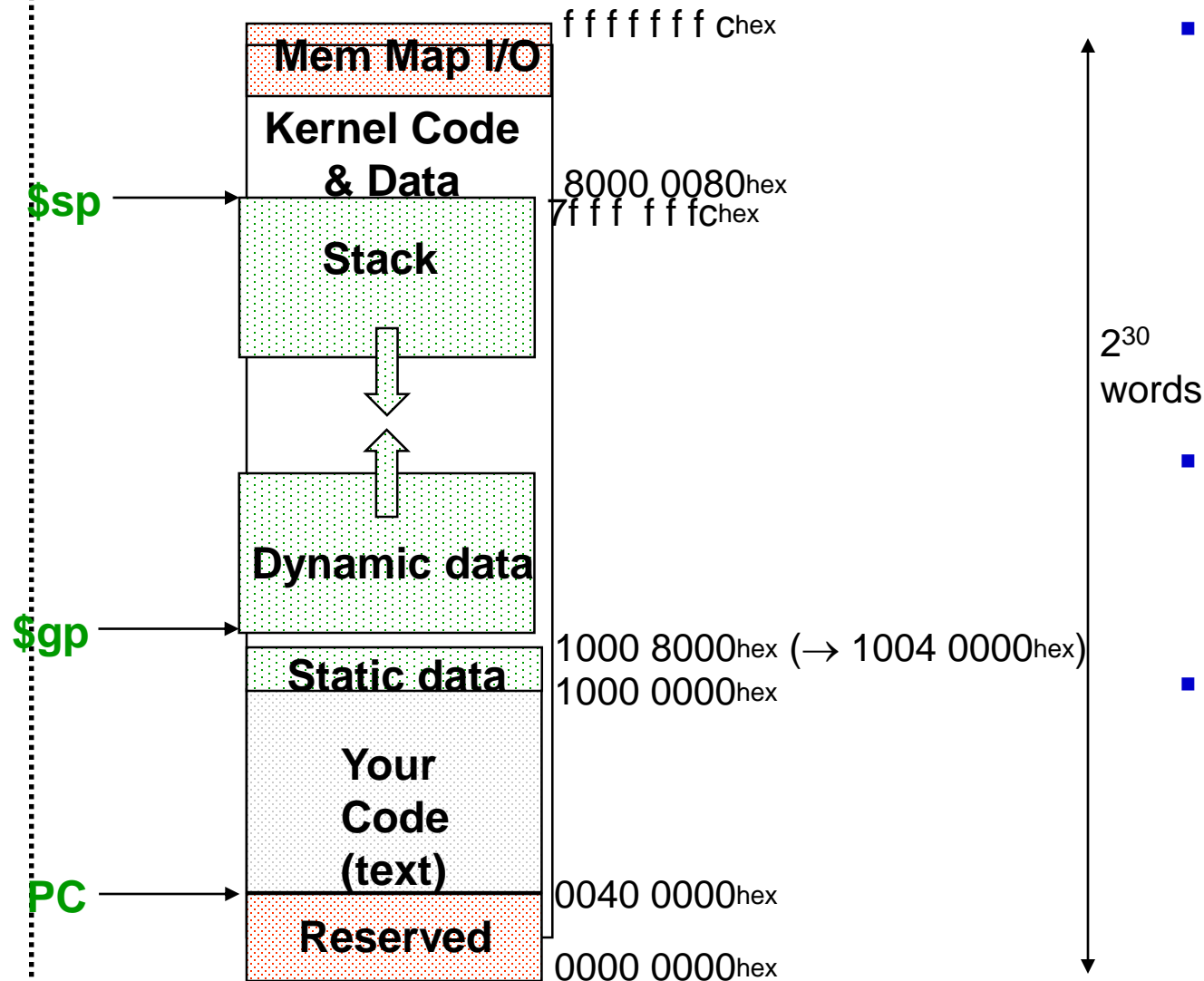


Other pointers

- Beside storing the saved registers when a procedure is called, the stack also stores local arrays and structures that do not fit in the register file;
- The segment of stack where a procedure saves registers and variables local to the procedure that do not fit in registers is the “**procedure frame**.”
- **\$fp** is the “frame pointer” - points to the first word of a procedure frame
- **\$fp** allows a procedure to be called with more than 4 arguments – rest in memory.
- Compilers save time since **\$fp** is stable (it holds the address of $\$sp$ when initiated – obviates the need for arithmetic ops to restore $\$sp$



MIPS Memory Allocation (hex addresses)



- Static variables are not specific to a procedure. MIPS uses a global pointer **\$gp**.
- The machine code (text) is also stored in memory.
- The bottom of the memory space is reserved for the operating system.

MIPS Register Convention

Name	Register Number	Usage	Should <i>preserve</i> on call?
\$zero	0	the constant 0	no
\$v0 - \$v1	2-3	returned values	no
\$a0 - \$a3	4-7	arguments	yes
\$t0 - \$t7	8-15	temporaries	no
\$s0 - \$s7	16-23	saved values	yes
\$t8 - \$t9	24-25	temporaries	no
\$gp	28	global pointer	yes
\$sp	29	stack pointer	yes
\$fp	30	frame pointer	yes
\$ra	31	return address	yes

Register 1 \$at for assembler, Reg. 26-27 \$k0, \$k1 for OS

MIPS Data Types



Bit String: sequence of bits of a particular length

8 bits is a byte

32 bits (4 bytes) is a word

64 bits is a double-word

Integer: (signed or unsigned) 32 bits

Character: 8 bits – a byte can represent a character
(ASCII convention)

Floating point numbers: 32 bits

Floating point double precision: 64 bits

Memory addresses (pointers): 32 bits

Instructions: 32 bits



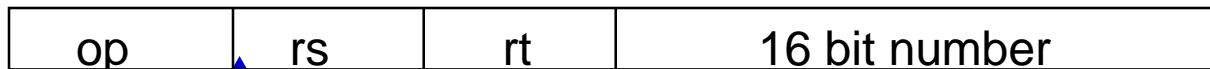
MIPS Character representation

- Computers use 8-bit bytes to represent characters with the **A**merican **S**tandard **C**ode for **I**nfo **I**nterchange (ASCII)

ASCII	Char		ASCII	Char	ASCII	Char	ASCII	Char		ASCII	Char	ASCII	Char
0	Null		32	space	48	0	64	@		96	`	112	p
1			33	!	49	1	65	A		97	a	113	q
2			34	“	50	2	66	B		98	b	114	r
3			35	#	51	3	67	C		99	c	115	s
4	EOT		36	\$	52	4	68	D		100	d	116	t
5			37	%	53	5	69	E		101	e	117	u
6	ACK		38	&	54	6	70	F		102	f	118	v
7			39	‘	55	7	71	G		103	g	119	w
8	bksp		40	(56	8	72	H		104	h	120	x
9	tab		41)	57	9	73	I		105	i	121	y
10	LF		42	*	58	:	74	J		106	j	122	z
11			43	+	59	;	75	K		107	k	123	{
12	FF		44	,	60	<	76	L		108	l	124	
15			47	/	63	?	79	O		111	o	127	DEL

Loading and Storing Bytes

- MIPS provides special instructions to move bytes
 - lb** `$t0, 101($s3) #load byte from memory`
 - lb places the byte from memory in the *rightmost* 8 bits of the destination register `$t0` and *sign extends it*
 - lbu** `$t0, 101($s3) #load byte from memory`
 - “load byte unsigned” lbu takes the byte from memory and places it in the rightmost 8 bits of the destination register `$t0` without sign extending it
 - sb** `$t0, 101($s3) #store byte to memory`
 - sb takes the byte from the *rightmost* 8 bits of register `$t0` and writes it to a byte in memory



A string copy leaf procedure

- The characters in the string are each represented by a number (according to the ASCII convention).
- A string copy means doing repeated **lb** and then **sb** to save the copied string, one character at a time.
- By convention, the end of a string is marked by a byte whose value is 0 (00000000)
- So the string "Grade" will be 71, 114, 97, 100, 101, 0
- The leaf procedure `strcpy` copies the string of characters `y[i]` into the string of characters `x[i]`

```
void strcpy (char x[], char y[])
{
  int i;
  i=0;
  while ((x[i]=y[i]) != '\0')
    i+=1;
}
```

MIPS code for string copy

strcpy:

```
add $t0 , $zero, $zero #i=0
```

```
L1: add $t1, $t0, $a1 #assume base address of  
    y[i] is passed in argument register $a1
```

```
lb $t2, 0($t1)#lower 8 bits of $t2=y[i]
```

```
add $t3, $t0, $a0 #assume base address of  
    x[i] is passed in argument register $a0
```

```
sb $t2, 0($t3)#x[i]=y[i]
```

```
beq $t2, $zero, L2#if y[i]==0 go to L2
```

```
addi $t1, $t1, 1 #i=i+1
```

```
j L1
```

```
L2: jr $ra # return from leaf procedure
```

MIPS Instructions, so far

Category	Instruction	Op Code	Example	Meaning
Arithmetic (<i>R</i> format)	add	0	add \$s1, \$s2, \$s3	$\$s1 = \$s2 + \$s3$
	subtract	0	sub \$s1, \$s2, \$s3	$\$s1 = \$s2 - \$s3$
Data transfer (<i>I</i> format)	load word	35	lw \$s1, 100(\$s2)	$\$s1 = \text{Memory}(\$s2+100)$
	store word	43	sw \$s1, 100(\$s2)	$\text{Memory}(\$s2+100) = \$s1$
	load byte	32	lb \$s1, 101(\$s2)	$\$s1 = \text{Memory}(\$s2+101)$
	store byte	40	sb \$s1, 101(\$s2)	$\text{Memory}(\$s2+101) = \$s1$
	load byte unsigned	36	lbu \$t0, 101(\$s2)	$\$t0 = \text{Memory}(\$s2+101)$

What About Larger Constants?

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

lui \$t1, 257

15	0	9	256
----	---	---	-----



001111	00000	01001	0000 0001 0000 0001
--------	-------	-------	---------------------

- After the lui operation the register \$t1 looks like

0000 0001 0000 0001	0000 0000 0000 0000
---------------------	---------------------



Loading 32-bit constants in a register (part 2)

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

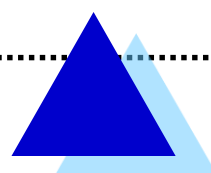
```
ori $t1, $t1, 2305
```

3	9	9	2305
---	---	---	------

0000 0000 0000 0000	0000 1001 0000 0001
---------------------	---------------------

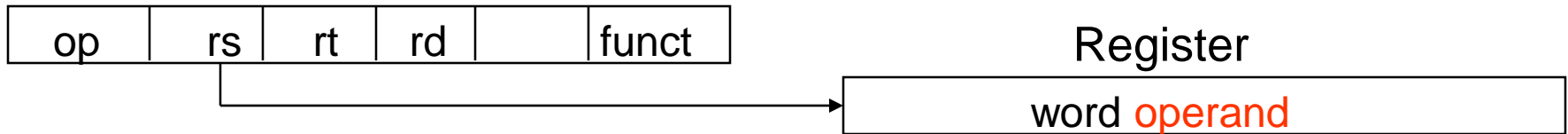
0000 0001 0000 0001	0000 0000 0000 0000
---------------------	---------------------

0000 0001 0000 0001	0000 1001 0000 0001
---------------------	---------------------

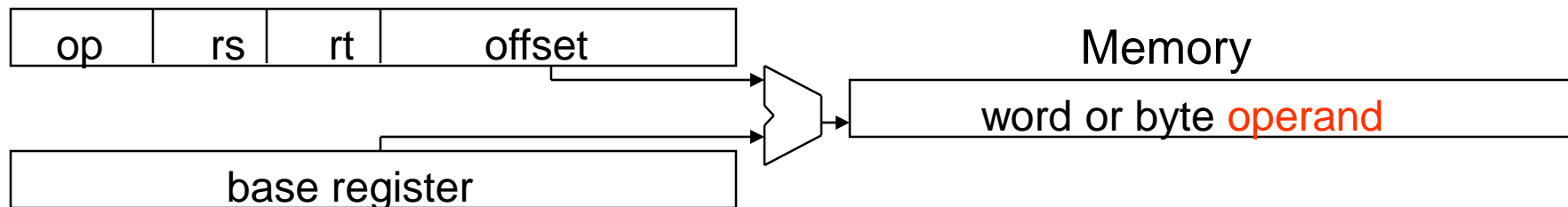


MIPS Addressing Modes

- *Register addressing* – **operand** is in a register

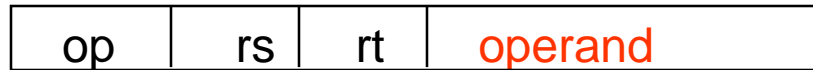


- *Base (displacement) addressing* – **operand** is at the memory location whose address is the sum of a register and a 16-bit constant contained within the instruction

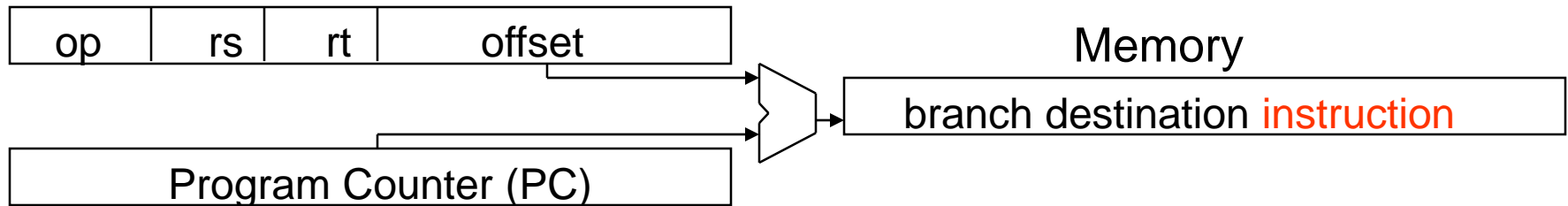


MIPS Addressing Modes

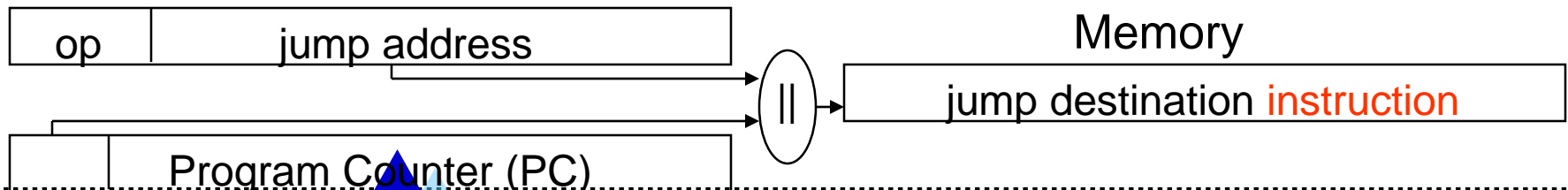
- Immediate addressing – **operand** is a 16-bit constant contained within the instruction



- PC-relative addressing – instruction **address** is the sum of the PC and a 16-bit constant within the instruction

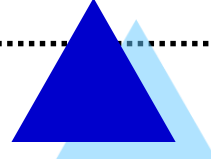


- Pseudo-direct addressing – instruction **address** is the 26-bit constant contained within the instruction concatenated with 00 LSBs and the upper 4 bits of the PC (MSBs)





Synchronization of shared memory locations

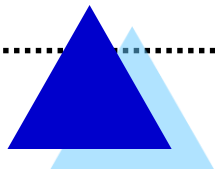
- ◆ Two processors sharing an area of memory
 - P1 writes, then P2 reads
 - Data race if P1 and P2 don't synchronize
 - ◆ Result depends of order of accesses
 - ◆ Hardware support required
 - Atomic read/write memory operation
 - No other access to the location allowed between the read and write
 - ◆ Could be a single instruction
 - E.g., atomic swap of register \leftrightarrow memory
 - Or an atomic pair of instructions
- 



Synchronization of shared memory locations

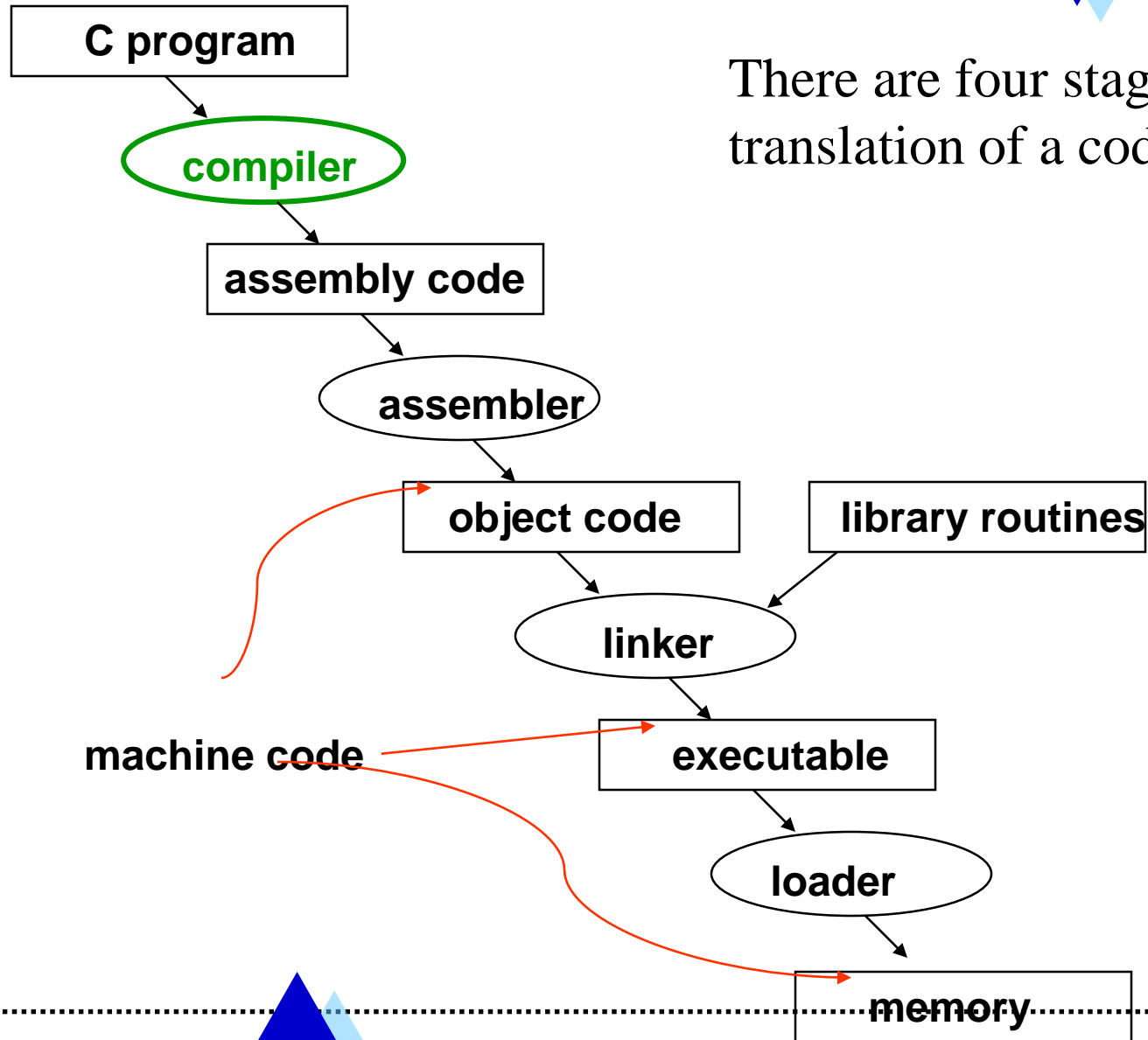
- ◆ Load linked: **ll** rt, offset(\$s1)
- ◆ Store conditional: **sc** rt, offset(\$s1)
 - Succeeds if location not changed since the **ll**
 - ◆ Returns 1 in rt
 - Fails if location is changed
 - ◆ Returns 0 in rt
- ◆ Example: atomic swap (to test/set lock variable)

```
try: add $t0,$zero,$s4 ;copy exchange value
      ll  $t1,0($s1)    ;load linked
      sc  $t0,0($s1)    ;store conditional
      beq $t0,$zero,try ;branch store fails
      add $s4,$zero,$t1 ;put load value in $s4
```

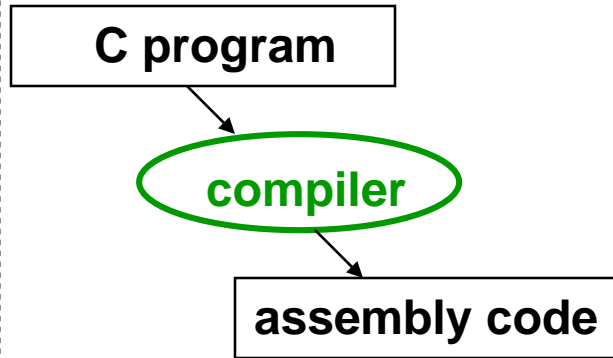


The Code Translation Hierarchy

There are four stages in the translation of a code



Compiler

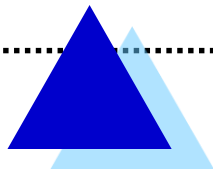


- Transforms the C program into an assembly language program
- Advantages of high-level languages
 - many fewer lines of code
 - easier to understand and debug
 - Portability across platforms
- Today's optimizing compilers can produce assembly code nearly as good as an assembly language programming expert and often better for **large** programs



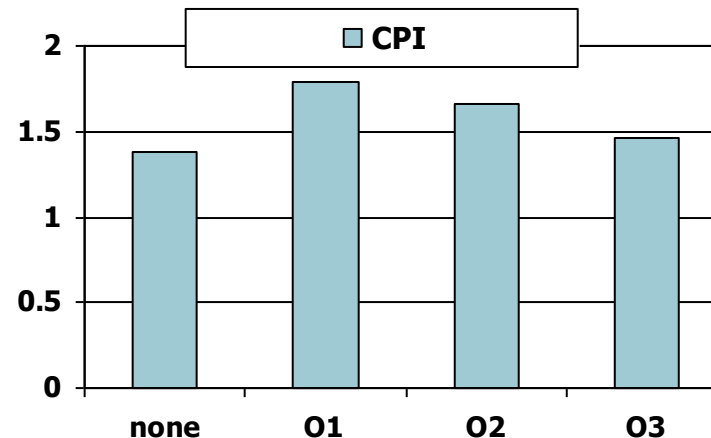
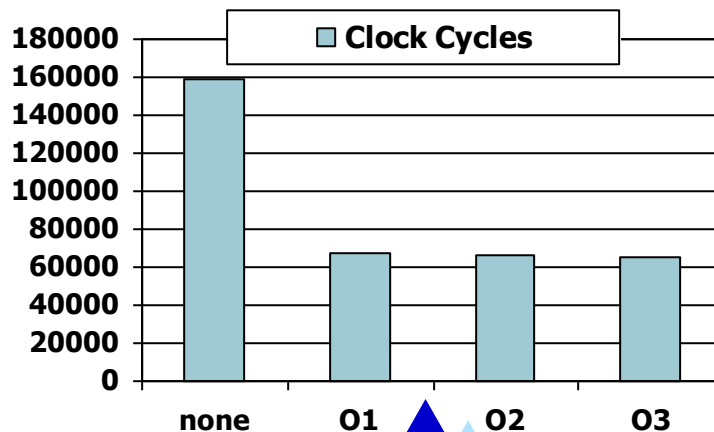
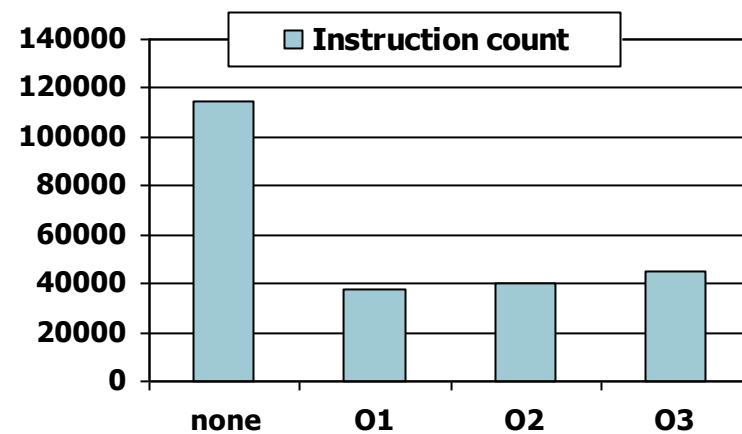
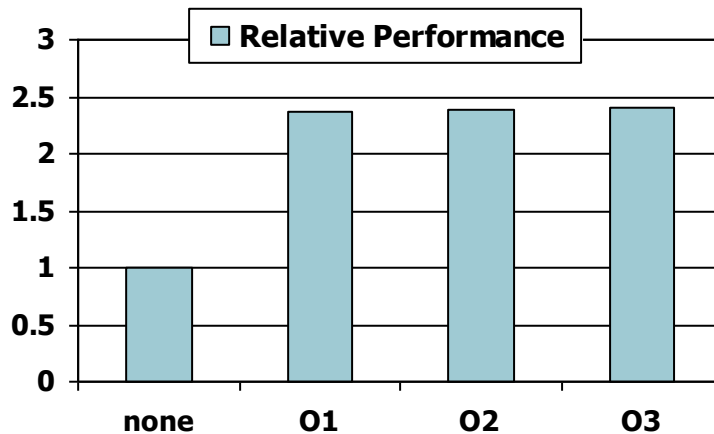
Types of Compiler Optimization

Gcc optimization	Optimization Name	Explanation
01 (medium)	Local	Processor dependent, reorder instructions to improve pipeline performance; replace multiply by constant with shifts; replace two instances of a same computation with a single copy
02 (full)	Global	Remove code from loop that computes same value at each iteration, eliminate array addressing within loops
03	High level	Processor independent – near source level, replace procedure call by procedure body

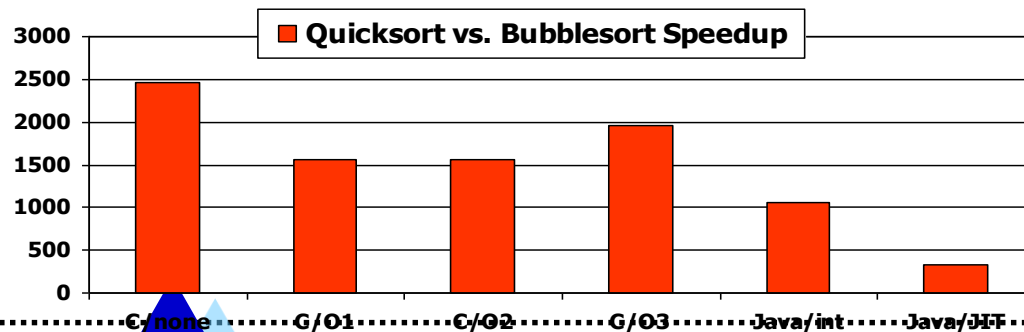
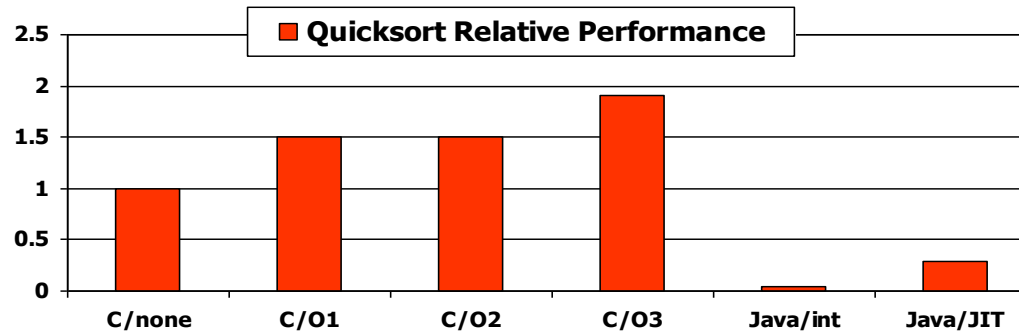
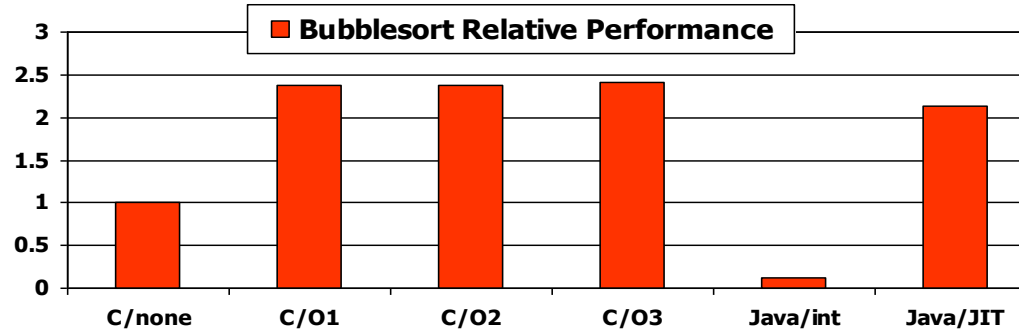


Impact of Compiler Optimization

Compiled with gcc for Pentium 4 under Linux

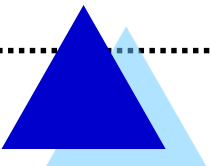


Impact of Compiler Optimization

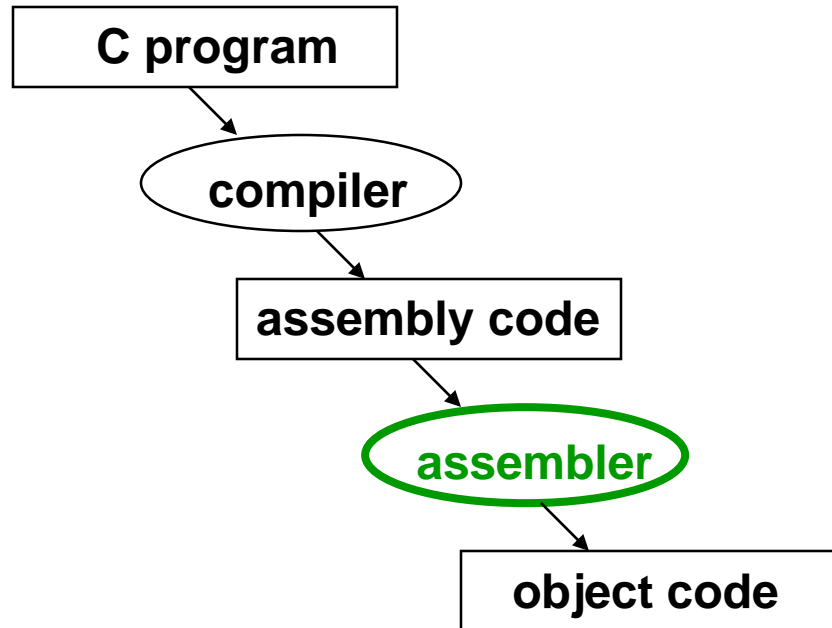




Impact of Compiler Optimization

- ◆ Instruction count and CPI are not good performance indicators in isolation
 - ◆ Compiler optimizations are sensitive to the algorithm
 - ◆ Java/JIT compiled code is significantly faster than JVM interpreted
 - Comparable to optimized C in some cases
 - ◆ Nothing can fix a dumb algorithm!
- 

The Code Translation Hierarchy



Assembler

- Transforms symbolic assembler code into object (machine) code
- Advantages of assembly language
 - ✓ Programmer has more control compared to higher level language
 - ✓ much easier than remembering instruction binary codes
 - ✓ can use labels for addresses – and let the assembler do the arithmetic
 - ✓ The frequency of instructions is about 50% arithmetic (`add`, `addi`, `sub`), 40% data transfer (`lw`, `sw`, `lb`, `sb`, `lui`), 8% conditional branches (`beq`, `bne`, `slt`, `slti`) and 2% jumps (`j`, `jr`, `jal`)
- The assembler also understands pseudo-instructions. It transforms the pseudo-instruction into correct machine language code.

Assembler

“**move** \$t0, \$t1” exists only in assembler – it is implemented using

add \$t0, \$t1, \$zero

op	rs	rt	rd	shift	funct
----	----	----	----	-------	-------

R-type

0	9	0	8	0	32
---	---	---	---	---	----

R-type

- When considering performance, you should count **real** instructions **executed**, not code size (which includes pseudo-instructions). This is because some pseudo-instructions generate several assembler instructions:

ble \$t5, \$t3, Label #go to Label if \$t5 ≤ \$t3

Is slt \$at, \$t3, \$t5 # \$at=1 if \$t3 < \$t5

beq \$at, \$zero, Label #go to Label if \$at=0

Other Tasks of the Assembler

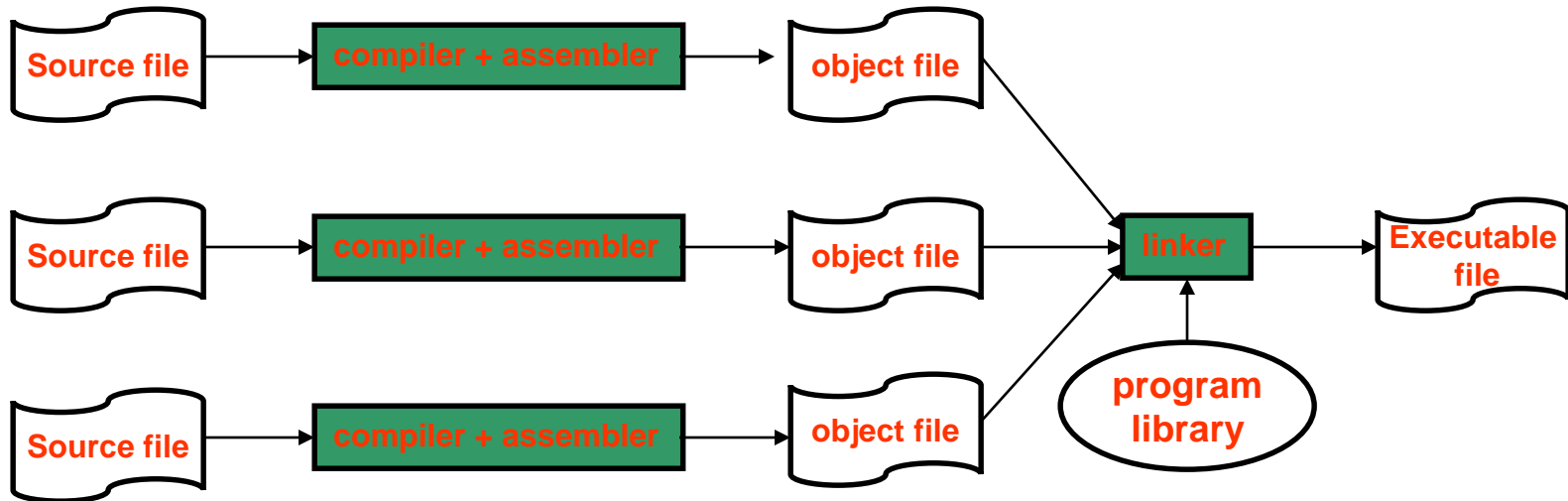
- Determines binary addresses corresponding to all labels
 - keeps track of labels used in branches and data transfer instructions in a **symbol table**
 - pairs of symbols and addresses
- Converts pseudo-instructions to legal assembly code
 - register **\$at** is reserved for the assembler to do this
- Converts branches to far away locations into a branch followed by a jump
- Converts instructions with large immediates into a **lui** (load upper immediate) followed by an **ori** (or immediate)
- Converts numbers specified in decimal and hexadecimal into their binary equivalents. Example
- $C08E_{\text{hex}} = 1100\ 0000\ 1000\ 1110_2$
- Converts characters into their ASCII equivalents

Typical Object File Pieces

- Object file header: size and position of following pieces
- Text module: assembled object (machine) code
- Data module: data accompanying the code
 - static data - allocated throughout the program
 - dynamic data - grows and shrinks as needed by the program
- Relocation information: identifies instructions (data) that use (are located at) **absolute addresses** – those that are not relative to a register (e.g., jump destination addr) – when the code and data is loaded into memory
- Symbol table: remaining undefined labels (e.g., external references) used in branch and data transfers
- Debugging information

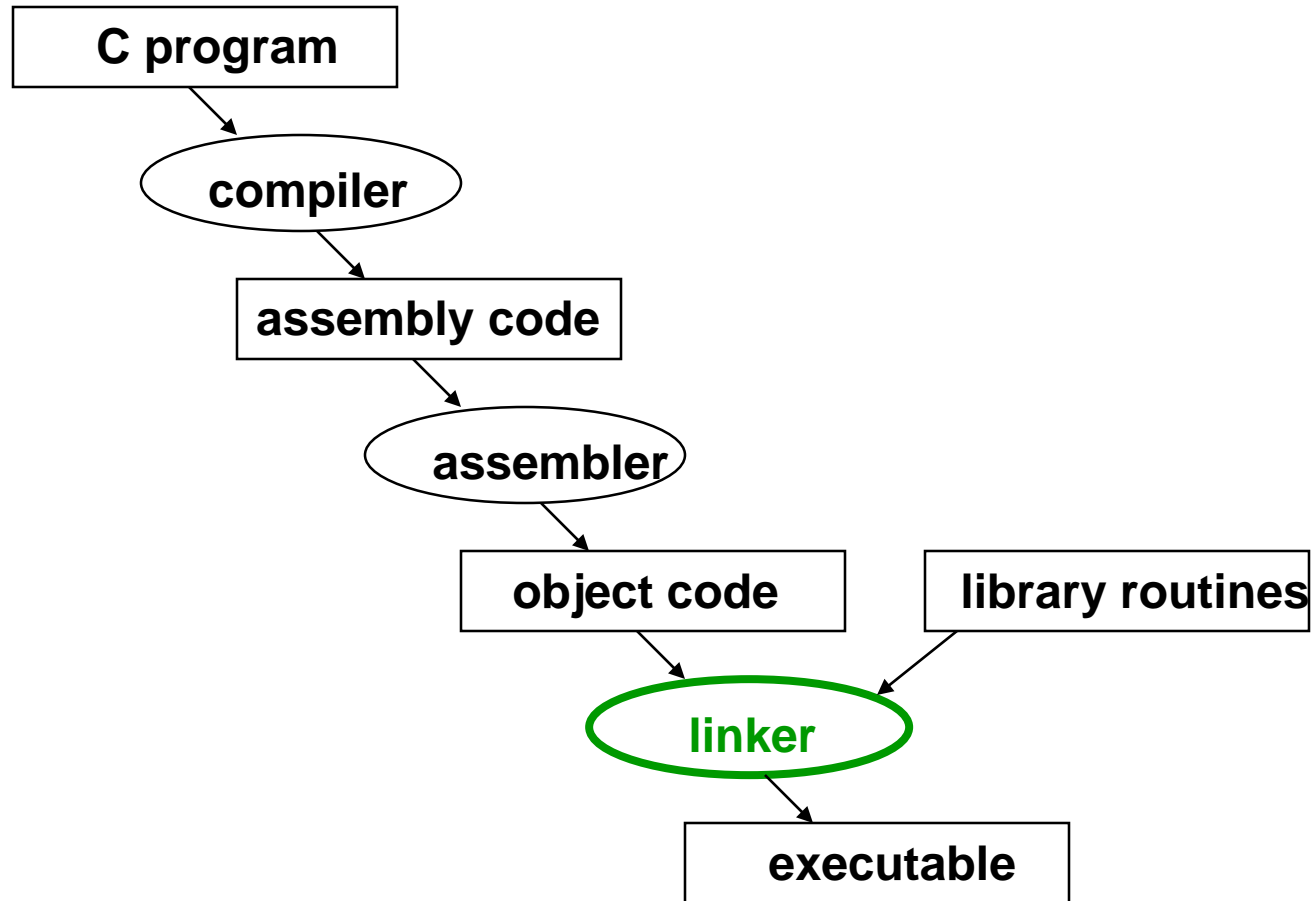


Process that produces an executable file



- Each program is compiled independently – a change in one program does not require compiling all others.

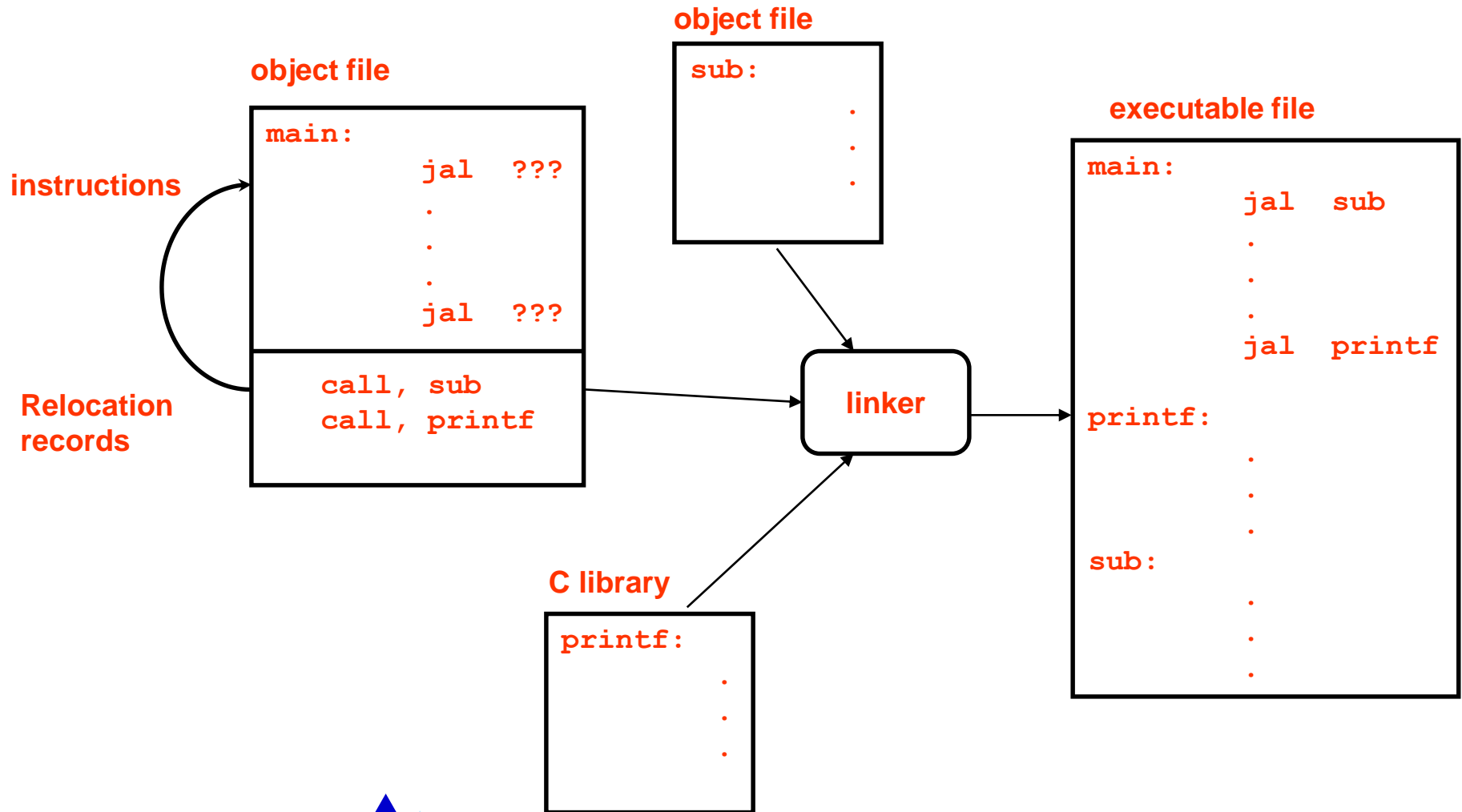
The Code Translation Hierarchy



Linker

- Takes all of the independently assembled code segments and “stitches” (links) them together
 - Much faster to patch code and recompile and reassemble that patched routine, than it is to recompile and reassemble the entire program
- Decides on memory allocation pattern for the code and data modules of each segment
 - remember, segments were assembled in isolation so **each** assumes its code’s starting location is 0x0040 0000 and its static data starting location is 0x1000 0000
- Absolute addresses must be **relocated** to reflect the new starting location of each code and data module
- Uses the symbol table information to resolve all remaining undefined labels
 - branches, jumps, and data addresses to external segments — *replaced with new addresses*

Linker



Linking example

Object file A

Name	Procedure A		
Text size	100hex		
Data size	20hex		
Text segment	0 lw \$a0, 0(\$gp)		
	4 jal 0		
		
Data segment	0 (X)		
		
Relocation info	0 lw	X	
	4 jal	B	
Symbol table	X	-	
	B	-	

address Instruction dependency

Object file B

Name	Procedure B		
Text size	200hex		
Data size	30hex		
Text segment	0 sw \$a0, 0(\$gp)		
	4 jal 0		
		
Data segment	0 (Y)		
		
Relocation info	0 sw	Y	
	4 jal	A	
Symbol table	Y	-	
	A	-	

Resulting executable file

Executable file header

Text size 300hex (100 hex for Procedure A and 200 hex for B)

Data size 50hex (20 hex for Procedure A and 30 hex for B)

Text segment 0040 0000hex lw \$a0, 8000hex(\$gp)
 0040 0004hex jal 0040 0100hex

.....

 0040 0100hex sw \$a1, 8020hex(\$gp)

 0040 0104hex jal 0040 0000hex

....

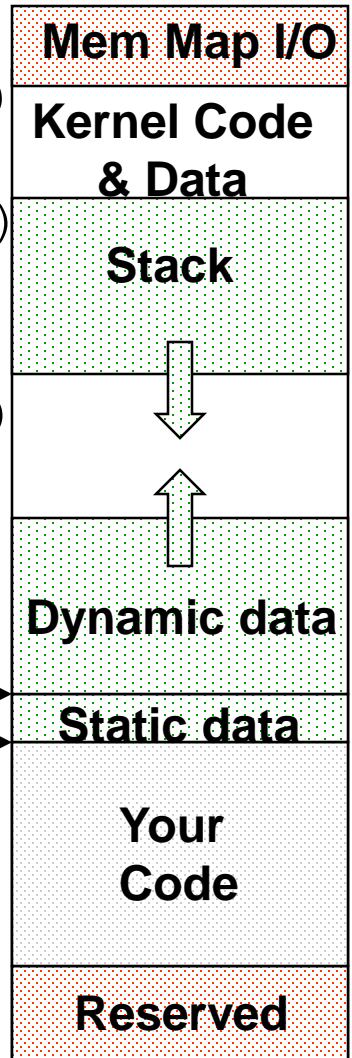
Data segment 1000 0000hex (X)

.....

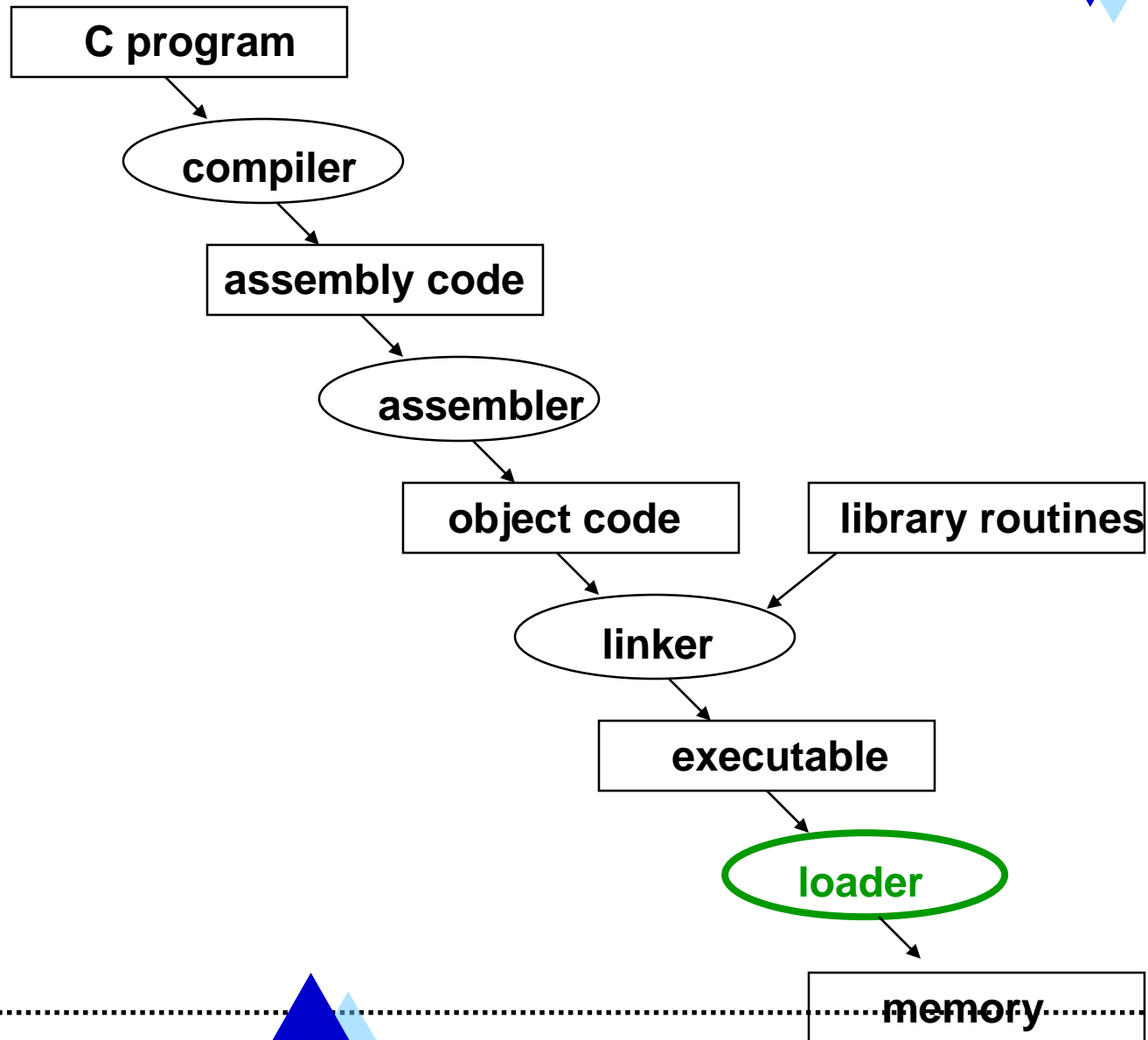
 1000 0020hex (Y)

....

PC 0040 0000hex →

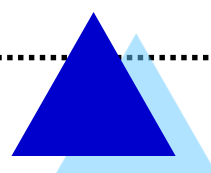


The Code Translation Hierarchy





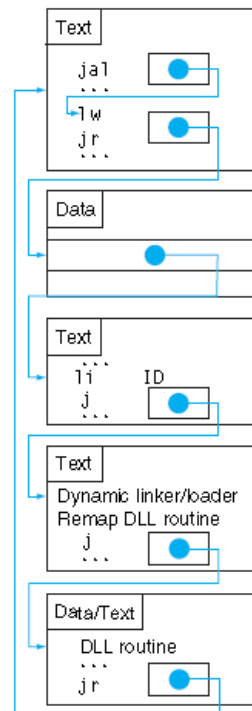
Loader

- Loads (copies) the executable code now stored on disk into memory (RAM) at the starting address specified by the **operating system**
 - Creates an address space large enough for text and data
 - Initializes the machine registers and sets the stack pointer to the first free location (0x7ffe fffc)
 - Copies the parameters (if any) to the main routine onto the stack
 - Jumps to a start-up routine (at PC addr 0x0040 0000 on xspim) that copies the parameters into the argument registers and then calls the main routine of the program with a `jal main`
 - Terminates with `exit`
- 

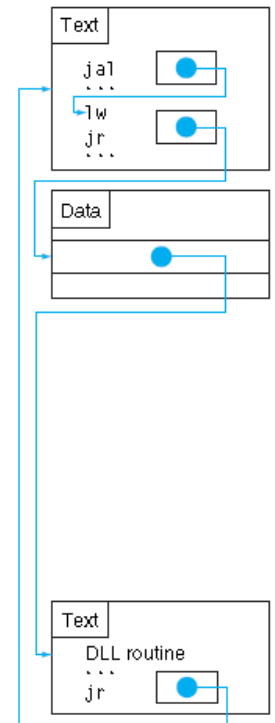
Dynamically linked libraries

- Useful when the size of the library is much larger than that of the program calling it.
- Modern OS (such as Windows) use dynamically linked libraries, such that each routine is linked only after it is called.
- The price to pay is more overhead the first time the routine is called

First time DLL is called



Subsequent DLL calls



Arrays vs. pointers



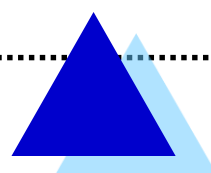
- Another way to optimize code is to use pointers instead of arrays. Consider two C-code routines that clear a sequence of words (array) in memory

```
clear1(int array[], int size)
```

```
{  
    int i;  
    for(i=0; i<size; i=i+1)  
        array[i]=0;  
}
```

```
clear2(int *array, int size)
```

```
{  
    int *p;  
    for(p=&array[0]; p<&array[size]; p=p+1)  
        *p=0;  
}
```



Arrays vs. pointers

- We know that `array` base address is passed in `$a0`, `size` in `$a1`. We assume `$t0` holds the array index `i`. When using pointers, the pointer `p` is stored in `$t0`.
- Version without pointers

```
move $t0,$zero #i=0
L1: add $t1, $t0, $t0 # $t1=2*i
    add $t1, $t1, $t1 # $t1=4*i
    add $t2,$a0, $t1 # $t2 address of array[i]
    sw $zero, 0($t2) # array[i]=0
    addi $t0, $t0, 1 # i=i+1
    slt $t3, $t0, $a1 # $t3=1 if i<size
    bne $t3, $zero, L1 # if i<size go to L1
```

7 instructions executed for every loop.

Arrays vs. pointers



- the version using pointers

```
move $t0, $a0 #p=address of array[0]
add $t1, $a1, $a1 # $t1=2*size
add $t1, $t1, $t1 # $t1=4*size
add $t2, $a0, $t1 # $t2 address of array[size]
L2: sw $zero, 0($t0) #memory[p]=0
addi $t0, $t0, 4 #p=p+4
slt $t3, $t0, $t2 # $t3=1 if p<address of array[size]
bne $t3, $zero, L2 #if p<address of array[size] go to L2
```

4 instructions executed for every loop.

