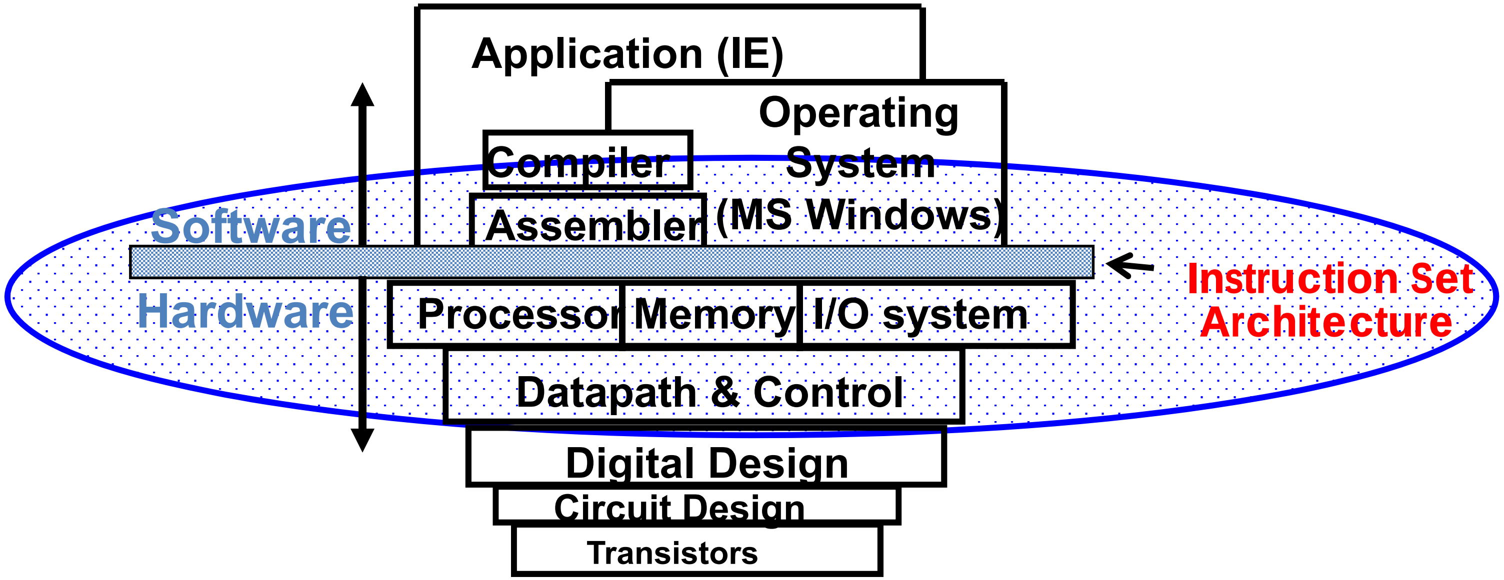
Chapter 2

Instructions: Language of the Computer

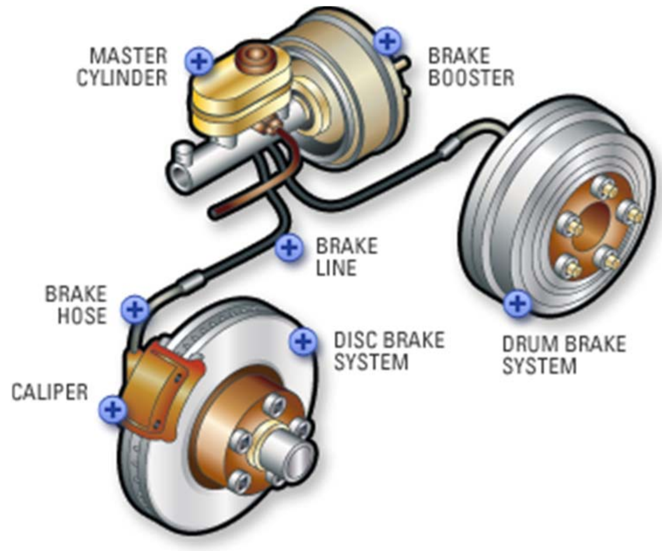
Instruction Set

- Instruction Set : set of instructions of a computer
- Different computers have different instruction sets
  - But with many aspects in common
- Interface between hardware and software of a computer



When to use Assembly Language

- The speed or size of a program is critically important
  - For example, a computer control a car's brakes
  - Need to respond rapidly and predictably to events in the outside world
  - High-level languages



- The ability to exploit specialized instruction
  - For example, string copy or pattern-matching instructions

The MIPS Instruction Set

- Stanford MIPS commercialized by MIPS Technologies (www.mips.com)
  - Used as the example throughout the book
- Large share of embedded core market
  - Applications in consumer electronics, network/storage equipment, cameras, printers, …
- Typical of many modern ISAs
  - Different assembly languages quite similar
  - Knowing MIPS will help learn new assembly languages
  - See MIPS Reference Data tear-out card, and Appendixes B and E

Instructions in Chapter 2

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	$s1 = $s2 + $s3	Three register operands
	subtract	`sub $s1,$s2,$s3`	$s1 = $s2 – $s3	Three register operands
	add immediate	`addi $s1,$s2,20`	$s1 = $s2 + 20	Used to add constants
Data transfer	load word	`lw $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	`sw $s1,20($s2)`	Memory[$s2 + 20] = $s1	Word from register to memory
	load half	`lh $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	load half unsigned	`lhu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	`sh $s1,20($s2)`	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte	`lb $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	load byte unsigned	`lbu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	`sb $s1,20($s2)`	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	`ll $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	`sc $s1,20($s2)`	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	`lui $s1,20`	$s1 = 20 * 2^{16}	Loads constant in upper 16 bits
Logical	and	`and $s1,$s2,$s3`	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	`or $s1,$s2,$s3`	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	`nor $s1,$s2,$s3`	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	`andi $s1,$s2,20`	$s1 = $s2 & 20	Bit-by-bit AND reg with constant
	or immediate	`ori $s1,$s2,20`	$s1 = $s2 | 20	Bit-by-bit OR reg with constant
	shift left logical	`sll $s1,$s2,10`	$s1 = $s2 << 10	Shift left by constant
	shift right logical	`srl $s1,$s2,10`	$s1 = $s2 >> 10	Shift right by constant
Conditional branch	branch on equal	`beq $s1,$s2,25`	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1,$s2,25`	if ($s1!= $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for beq, bne
	set on less than unsigned	`sltu $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than unsigned
	set less than immediate	`slti $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant
	set less than immediate unsigned	`sltiu $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant unsigned
Unconditional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to $ra	For switch, procedure return
	jump and link	`jal 2500`	$ra = PC + 4; go to 10000	For procedure call

Arithmetic Operations

- Add and subtract, three operands
  - Two sources and one destination

```
add a, b, c  # a=b + c
sub a, b, c  # a=b - c
```

- All arithmetic operations have this form
- *Design Principle 1:* Simplicity favors regularity
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

`Words after # are comment for humman`

Arithmetic Example

- C code:

```
f = (g + h) - (i + j);
```

- Compiled MIPS code:

```
add t0, g, h   # temp t0 = g + h
add t1, i, j   # temp t1 = i + j
sub f, t0, t1  # f = t0 - t1
```

Register Operands

- Arithmetic instructions use register operands
- MIPS has a 32 × 32-bit register file
  - Use for frequently accessed data
  - Registers numbered 0 to 31
  - 32-bit data called a "word"
- *Design Principle 2:* Smaller is faster
  - Smaller register file make operation fast
  - Much faster than main memory (which has millions of locations)

Naming Conventions for Register

- $t0, $t1....$t9 for temporary values
- $s0, $s1,....$s7 for saved variable
- Register 1, called $at, is reserved for the assembler (see Section 2.12),

Name	Register number	Usage	Preserved on call?
$zero	0	The constant value 0	n.a.
$v0–$v1	2–3	Values for results and expression evaluation	no
$a0–$a3	4–7	Arguments	no
$t0–$t7	8–15	Temporaries	no
$s0–$s7	16–23	Saved	yes
$t8–$t9	24–25	More temporaries	no
$gp	28	Global pointer	yes
$sp	29	Stack pointer	yes
$fp	30	Frame pointer	yes
$ra	31	Return address	yes

Register Operand Example

- Find the compiled MIPS code for the following C code:

```
f = (g + h) - (i + j);
```

$s0	f
$s1	g
$s2	h
$s3	i
$s4	j

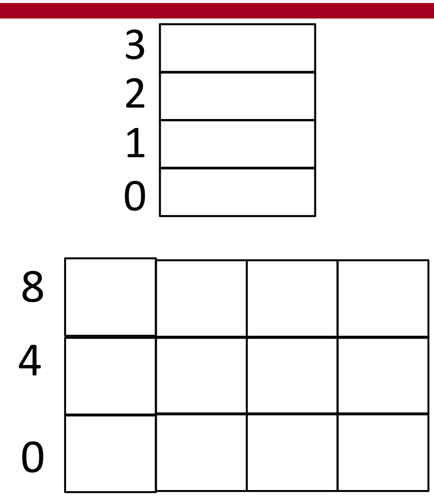
```
add $t0, $s1, $s2  # $t0 = g+h
add $t1, $s3, $s4  # $t1 = i+j
sub $s0, $t0, $t1  # f = (g+h) – (i+j)
```

Big and Little Endian

- Main memory used for composite data
  - Arrays, structures, dynamic data
- Memory is byte addressable
  - Each address identifies an 8-bit byte
- Words are aligned in memory
  - Address must be a multiple of 4
- Big Endian: Most-significant byte (MSB) at least address of a word
- Little Endian: MSB at largest address
- For example, how the data is 0x12FE34DC stored in

MSB LSB

1. big endian
2. little endian



Aligned word

Address

Address

12	FE	34	DC	
0	1	2	3	big endian
3	2	1	0	little endian

Memory Operands

- MIPS is Big Endian
  - Most-significant byte at least address of a word

- Load values from memory into registers.

```
lw $s1, 20($s2)
=> $s1=Mem[$s2+20]
```

Still has some problems, see next slide

  - $s2 the register that contains the base address of the memory array
  - 20 is the offset

- Store result from register to memory

```
sw $s1, 20($s2)
=> Mem[$s2+20] =$s1
```

Address	
3	LSB
2	
1	
0	MSB

Memory Operand Example 1

- C code:

```
g = h + A[8];
```

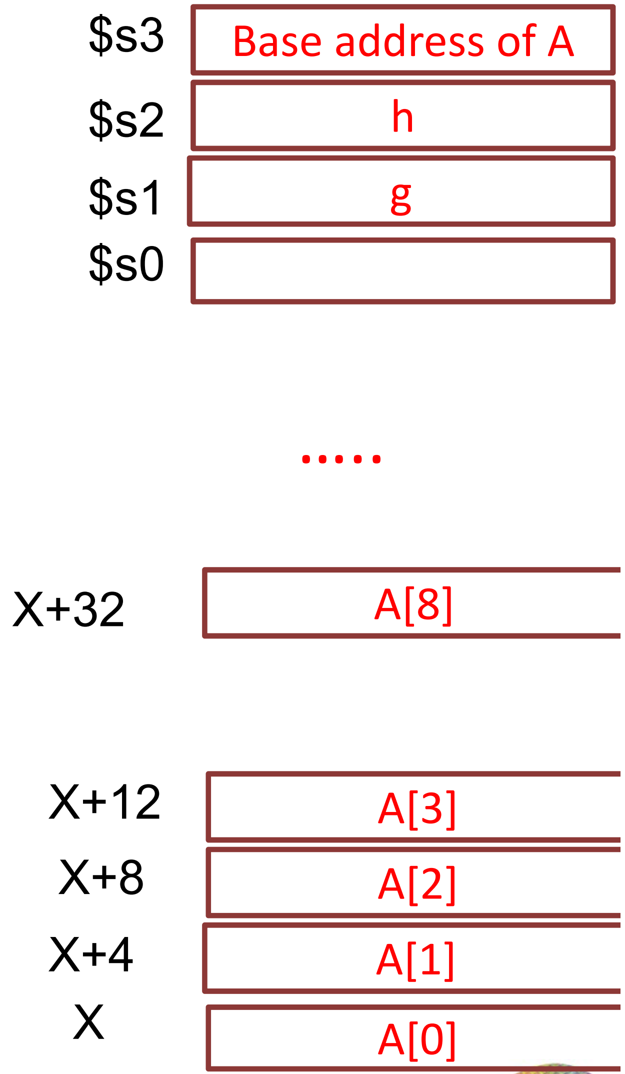
  - *g in $s1, h in $s2, base address of A in $s3*

- Compiled MIPS code:
  - Index 8 requires offset of 32
    - 4 bytes per word

```
lw $t0, 32($s3) #load word
add $s1, $s2, $t0
```

offset

base register



Hidden

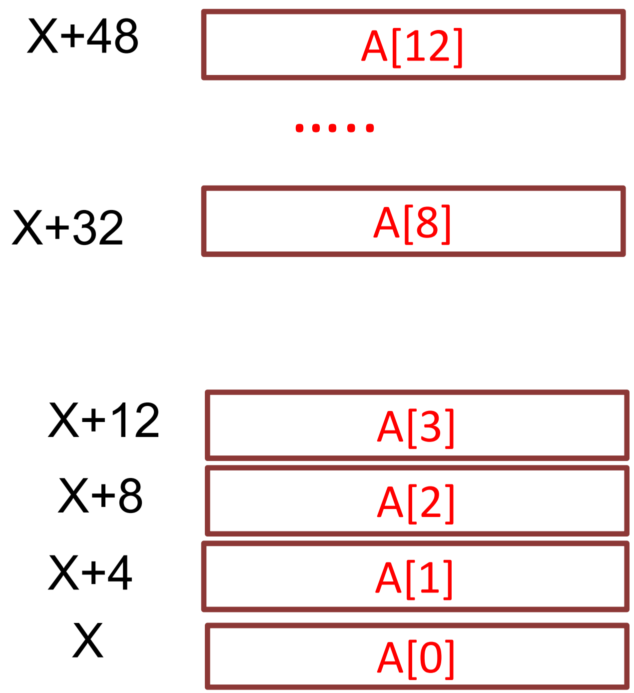
Memory Operand Example 2

- Convert the following C code to MIPS instruction

```
A[12] = h + A[8];
```

  – *h in $s2, base address of A in $s3*

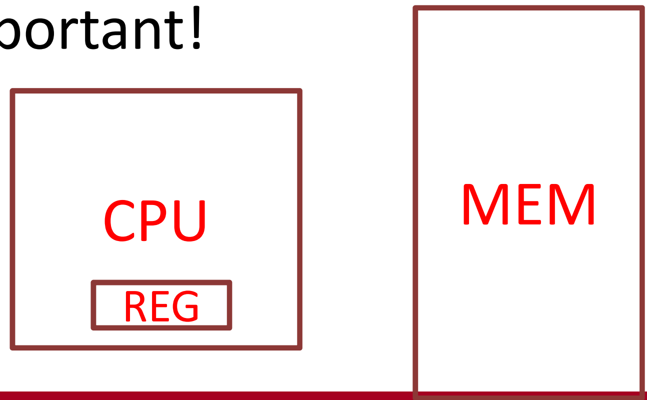
- Compiled MIPS code:
  – Index 8 requires offset of 32




```
lw $t0, 32($s3) #load word
add $t0, $s2, $t0  #$t0 is a temporary reg.
sw $t0, 48($s3)
```

Registers vs. Memory

- Registers are faster than memory (memory is much larger)
- Operating on memory data requires loads and stores
  - Data are loaded into registers before processed
  - More instructions to be executed
  - But no need to process register and memory data at the same time => simplified
- Compiler must use registers for variables as much as possible
  - Only put less frequently used variables to memory (spilling)
  - Register optimization is important!



Constant or Immediate Operands

- Constant data specified in an instruction

```
addi $s3, $s3, 1 => $s3=$s3+1
```

- No subtract immediate instruction
  - Just use a negative constant

Assume Mem[$s1+0] contains value -1

⇓

```
addi $s3, $s3, -1   ⟷   lw $s1, 0($s0)
                        add $s3, $s3, $s1
```

One instruction instead of two instructions

- *Design Principle 3:* Make the common case fast
  - Add 1 and subtract 1 are common
  - Small constants are common
  - Immediate operand avoids a load instruction

The Constant Zero

- MIPS register 0 ($zero) is the constant 0
  - Cannot be changed
- Useful for common operations
  - E.g. move between registers

```
add $t2, $s1, $zero # $t2=$s1
```

Unsigned Binary Integers

- Given an n-bit number

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \Lambda + x_1 2^1 + x_0 2^0$$

- Range: 0 to $+2^n - 1$
- Example
  - $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$
    $= 0 + \ldots + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$
    $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$
- Using 32 bits
  - 0 to +4,294,967,295

2s-Complement Signed Integers

- Given an n-bit number

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \Lambda + x_1 2^1 + x_0 2^0$$

- Example
  - $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
    $= -1\times2^{31} + 1\times2^{30} + \ldots + 1\times2^2 + 0\times2^1 + 0\times2^0$
    $= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$
- Range: -2^{n-1} to $+2^{n-1} - 1$
- Using 32 bits
  - –2,147,483,648 to +2,147,483,647

01	1
00	0
11	-1
10	-2

2s-Complement Signed Integers

- Bit 31 is sign bit
  - 1 for negative numbers
  - 0 for non-negative numbers
- $-(-2^{n-1})$ can't be represented
- Non-negative numbers have the same unsigned and 2s-complement representation
  - 5 is 101_2 in both unsigned and 2s-complement signed
- Some specific numbers
  - 0: 0000 0000 … 0000
  - –1: 1111 1111 … 1111
  - Most-negative: 1000 0000 … 0000
  - Most-positive: 0111 1111 … 1111

Signed Negation

- Convert n to –n => Complement and add 1
  - Complement means 1 → 0, 0 → 1
- Example: negate +2
  - $+2 = 0000\ 0000 \ldots 0010_2$
  - $-2 = 1111\ 1111 \ldots 1101_2 + 1$
    $= 1111\ 1111 \ldots 1110_2$
- Reason for this

$$\text{because } x + \overline{x} = 1111\ldots111_2 = -1$$

$$\overline{x} + 1 = -x$$

Sign Extension

- Representing a number using more bits, but still preserve the numeric value
- Signed values: Replicate the sign bit to the left

  Examples: 8-bit to 16-bit

  +2: 0000 0010 => 0000 0000 0000 0010

  –2: 1111 1110 => 1111 1111 1111 1110

- Unsigned values: extend with 0s

  Examples: 8-bit to 16-bit

  +2: 0000 0010 => 0000 0000 0000 0010

Why do we need sign extension?

- Because an instruction is 32-bit, the constant or address in the instruction is less than 32-bit.

- In order keep the same value when putting the data into register, the constant or address must be signed extended.

Wrong

1111111110111111 ⇒ 0000000000 1111111110111111

16-bit

32-bit register

⇒ 1111111111111 1111111110111111 Correct

Why do we need sign extension?

- The constant or address in a instruction is less than 32-bit.
- Need sign extension to keep the same value when putting the data into register

```
addi $s3, $s3, -1 # $s3=$s3-1
```

op	rs	rt	constant
6 bits	5 bits	5 bits	16 bits

- Sign extension used in MIPS instruction set
  - addi: extend immediate value
  - lb, lh: extend loaded byte/halfword (discussed later)
  - beq, bne: extend the displacement (discussed later)

Representing Instructions

- Instructions are encoded in binary
  - Called machine code
- MIPS instructions
  - Encoded as 32-bit instruction words
  - Not many different instruction types
    - Regularity
    - Easier to implement
- Register that are frequently used
  - $t0 – $t7 are reg's 8 – 15
  - $t8 – $t9 are reg's 24 – 25
  - $s0 – $s7 are reg's 16 – 23

MIPS R-format Instructions

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- Instruction fields
  - op: operation code (opcode) => indicate the operation
  - rs: first source register number
  - rt: second source register number
  - rd: destination register number
  - shamt: shift amount (00000 for now) => (in Section 2.6)
  - funct: function code (extends opcode)=>select the specific variant of the operation in the op field

R-format Example (add, and, ..etc.)

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Note:
$s1=r17
$s2=r18
$t0=r8

```
add $t0, $s1, $s2
```

No shift

op	$s1	$s2	$t0	0	add
0	17_{10}	18_{10}	8	0	32
000000	10001	10010	01000	00000	100000

$$00000010001100100100000000100000_2 = 02324020_{16}$$

Recap: Hexadecimal

- Base 16
  - Compact representation of bit strings
  - 4 bits per hex digit

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

- Example: eca8 6420
  - 1110 1100 1010 1000 0110 0100 0010 0000

MIPS I-format Instructions (lw, sw, addi,…,etc.)

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

- Immediate arithmetic and load/store instructions
  - rt: destination or source register number
  - Constant: -2^{15} to $+2^{15}-1$ because 4^{th} field is 16-bit
  - Address: offset added to base address in rs

`addi $s3, $s3, 1` $s3=$s3+1

op	rs	rt	constant

`lw $t0, 32($s3)` $t0=MEM[$s3+32]

op	rs	rt	address

Example

- Translate the following statement into binary code

Opcode: lw:35_{10}, sw:43_{10}
\$t0:r8, \$t1:r9

```
lw $t0, 16($t1)
```

op	rs	rt	Address offset
35	9	8	16

```
sw $t0, 16($t1)
```

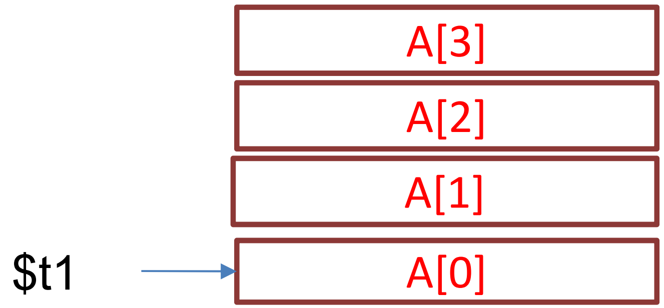
op	rs	rt	Address offset
43	9	8	16

Hidden

Example-2

- Translate the following statement into (1)MIPS instruction (2) binary code, assuming $t1 is the base address of A and $s2 contains h

A[3]=h+A[3]



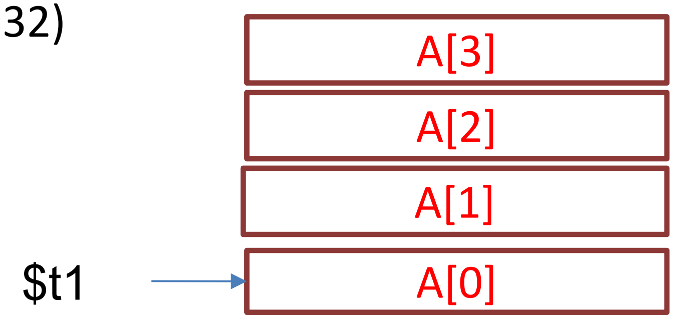
Example

- Translate the following statement into (1)MIPS instruction (2) binary code, assuming $t1 is the base address of A and $s2 contains h

Opcode:
lw:35, sw:43
Add: 0 (funct=32)
$t0:r8,
$s2: r18

```
A[3]=h+A[3]

lw $t0, 12($t1)
add $t0, $s2, $t0
sw $t0, 12($t1)
```



op	rs	rt	rd	shamt	funct
35	9	8	12		
0	18	8	8	0	32
43	9	8	12		

Instructions so far

MIPS machine language

Name	Format	Example						Comments
add	R	0	18	19	17	0	32	add $s1,$s2,$s3
sub	R	0	18	19	17	0	34	sub $s1,$s2,$s3
addi	I	8	18	17	100			addi $s1,$s2,100
lw	I	35	18	17	100			lw $s1,100($s2)
sw	I	43	18	17	100			sw $s1,100($s2)
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer format

FIGURE 2.6 MIPS architecture revealed through Section 2.5. The two MIPS instruction formats so far are R and I. The first16 bits are the same: both contain an *op* field, giving the base operation; an *rs* field, giving one of the sources; and the *rt* field, which specifies the other source operand, except for load word, where it specifies the destination register. R-format divides the last 16 bits into an rd field, specifying the destination register; the *shamt* field, which Section 2.6 explains; and the *funct* field, which specifies the specific operation of R-format instructions. I-format combines the last 16 bits into a single *address* field.

Logical Operations

- Instructions for bitwise manipulation

Operation	C	Java	MIPS
Shift left	<<	<<	`sll`
Shift right	>>	>>>	`srl`
Bitwise AND	&	&	`and, andi`
Bitwise OR	|	|	`or, ori`
Bitwise NOT	~	~	`nor`

- Useful for extracting and inserting groups of bits in a word

Shift Operations

- Shift left logical
  - Shift left and fill with 0 bits
  - `sll` by *i* bits multiplies by 2^i (00000011 << 2 => 00001100)
- Instruction format for sll: op:0, funct: 0
- shamt: how many positions to shift

```
Sll $t2, $s0, 2 # reg $t2= reg $s0 << 2bits
```

0	rs	rt	rd	shamt	0
6 bits	5 bits unused	5 bits	5 bits	5 bits	6 bits

0	0	16	10	2	0
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Shift Operations

- Shift right logical (srl)
  - Shift right and fill with 0 bits
  - `srl` by *i* bits divides by 2^i (unsigned only)
- Instruction format for srl: op:0, funct: 2
- shamt: how many positions to shift

0	rs	rt	rd	shamt	2
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

```
Srl $t2, $s0, 2 # reg $t2= reg $s0 >> 2bits
```

0	0	16	10	2	2
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

AND Operations

- Useful to mask bits in a word
  - Select some bits, clear others to 0

```
and $t0, $t1, $t2
```

\$t2	0000 0000 0000 0000 0000 1101 1100 0000
\$t1	0000 0000 0000 0000 0011 1100 0000 0000
\$t0	0000 0000 0000 0000 0000 1100 0000 0000

- Instruction format for and: op:0, funct: 100100_2

0	rs	rt	rd	shamt	100100_2
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

OR Operations

- Useful to include bits in a word
  - Set some bits to 1, leave others unchanged

```
or $t0, $t1, $t2
```

\$t2	0000 0000 0000 0000 0000 1101 1100 0000
\$t1	0000 0000 0000 0000 0011 1100 0000 0000
\$t0	0000 0000 0000 0000 0011 1101 1100 0000

- Instruction format for and: op:0, funct: 100101_2

0	rs	rt	rd	shamt	100101_2
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

NOT Operations

- Useful to invert bits in a word
  - Change 0 to 1, and 1 to 0
- MIPS uses NOR 3-operand instruction for NOT
  - a NOR b == NOT (a OR b)

```
not $t0,$t1 = nor $t0, $t1, $zero
```

Pseudo instruction

$t1	0000 0000 0000 0000 0011 1100 0000 0000
$t0	1111 1111 1111 1111 1100 0011 1111 1111

Conditional Operations

- Branch to a labeled instruction if a condition is true
  – Otherwise, continue sequentially

- `beq rs, rt, L1`
  – if (rs == rt) branch to instruction labeled L1;

- `bne rs, rt, L1`
  – if (rs != rt) branch to instruction labeled L1;

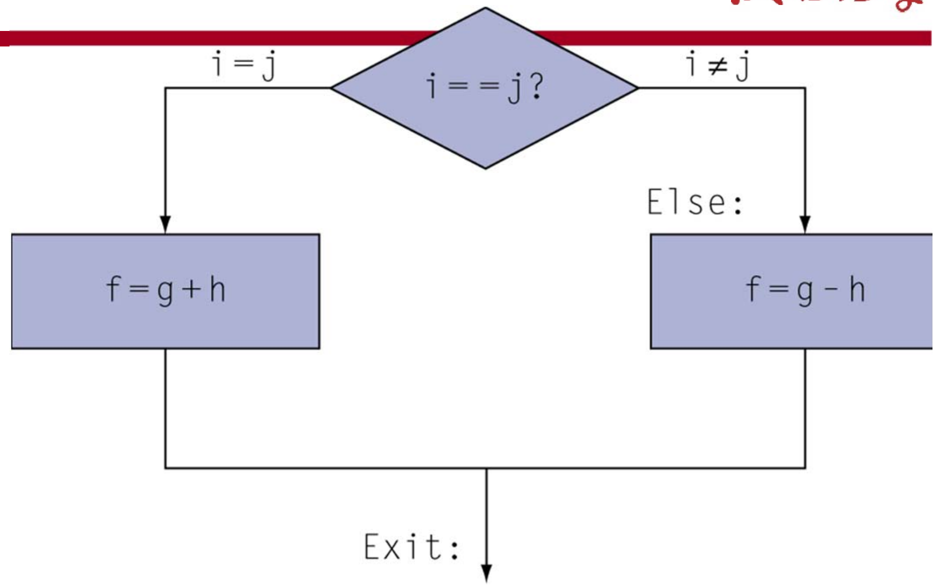
- `j L1`
  – unconditional jump to instruction labeled L1

Compiling If Statements

- C code:

```
if (i==j) f = g+h;
else f = g-h;
```

$s0	f
$s1	g
$s2	h
$s3	i
$s4	j

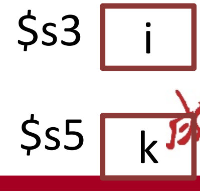


- Compiled MIPS code:

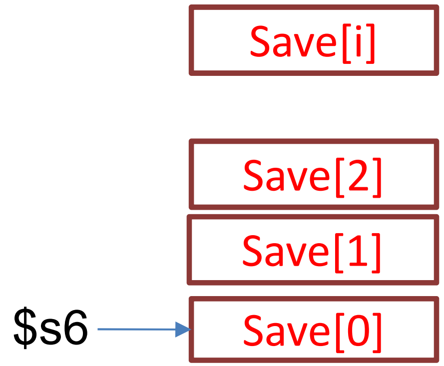
```
       bne $s3, $s4, Else   #if (i!=j) goto ELSE
       add $s0, $s1, $s2    # f= g+h
       j   Exit
Else:  sub $s0, $s1, $s2    # f= g-h
Exit:  …
```

Assembler calculates addresses

Compiling Loop Statements



- Convert the following C code to MIPS instruction, assume i is in $s3, k in $s5, base address of save is in $s6




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

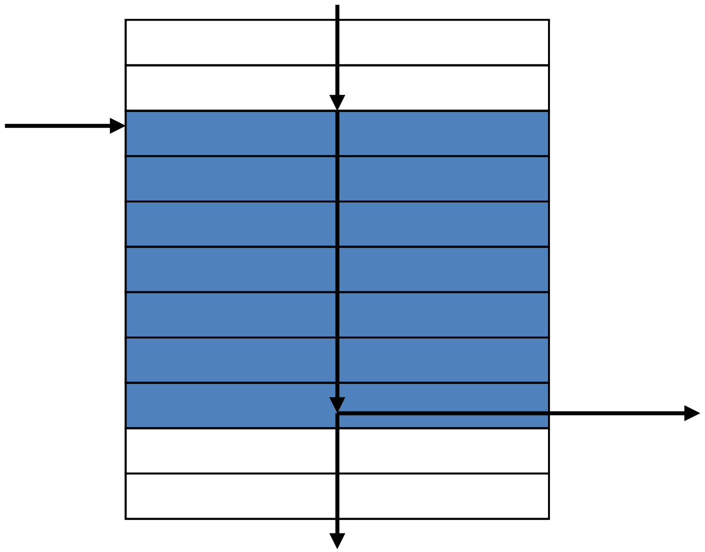
- Compiled MIPS code:

```
Loop: sll  $t1, $s3, 2    # $t1=i*4
      add  $t1, $t1, $s6  # address of save[i]
      lw   $t0, 0($t1)    # load save[i]
      bne  $t0, $s5, Exit # branch if save[i]!= k
      addi $s3, $s3, 1
      j    Loop
Exit: …
```

Basic Blocks

- A basic block is a sequence of instructions with
  - No embedded branches (except at end)
  - No branch targets (except at beginning)

```
Loop: sll  $t1, $s3, 2
      add  $t1, $t1, $s6
      lw   $t0, 0($t1)
      bne  $t0, $s5, Exit
      addi $s3, $s3, 1
      j    Loop
Exit: …
```



- A compiler identifies basic blocks for optimization
- An advanced processor can accelerate execution of basic blocks=>e.g. keep frequently data in registers

Conditional Operations: slt

- Set result to 1 if a condition is true
  - Otherwise, set to 0
- slt \$rd, \$rs, \$rt
  - if (\$rs < \$rt) \$rd = 1; else \$rd = 0;
- Use in combination with beq, bne to create relative conditions(equal, not equal, less than….)

```
if ($s1 < $s2)
branch to L
```

⇨

```
slt $t0, $s1, $s2
bne $t0, $zero, L
```

Demo (slt vs slti)

```
.data
   message: .asciiz "The number is less than the other"
.text
main:
   addi $t0, $zero, 1
   addi $t1, $zero, 200
   slt $s0, $t0, $t1
   bne $s0, $zero, printMessage

   li $v0, 10
   syscall
printMessage:
   li $v0, 4
   la $a0, message
   syscall
```

Output:
The number is less than the other

Because 1 < 200

https://www.youtube.com/watch?v=WF8jzQY0bh0&t=3s

Conditional Operations:slti

- slti rt, rs, constant //immediate mode
  - Set rt to 1 if (rs< constant)
  - Else rt =0
- Use in combination with beq, bne to create relative conditions(equal, not equal, less than….)

if ($s1 < 10)
branch to L

slti $t0, $s1, 10
bne $t0, $zero, L

Why not use a single instruction for blt (branch less than), bge (branch greater than), etc?

Demo (slt vs slti)

```
.data
   message: .asciiz "The number is less than the other"
.text
main:
   addi $t0, $zero, 1
   slti $s0, $t0, 10
   bne $s0, $zero, printMessage

   li $v0, 10
   syscall
printMessage:
   li $v0, 4
   la $a0, message
   syscall
```

Output:
The number is less than the other

Because 1 < 10

https://www.youtube.com/watch?v=WF8jzQY0bh0&t=3s

Why not blt and bgt

- because Hardware for $<, \leq, >, \geq$ are slower than $=, \neq$
  - $<, \leq, >, \geq$ use 2s-complement subtraction, but $=$ or $\neq$ use xor
  - Combining $<, \leq, >, \geq$ with branch involves more work per instruction, result in slower clock
  - All instructions penalized!
- beq and bne are the common case, no need to create instructions for blt and bgt
- => a good design compromise

Signed vs. Unsigned

- Signed integer comparison use: slt, slti
- Unsigned integer comparison use: sltu, sltiu
- Example
  - \$s0 = 1111 1111 1111 1111 1111 1111 1111 1111
  - \$s1 = 0000 0000 0000 0000 0000 0000 0000 0001

slt \$t0, \$s0, \$s1 # signed
\$t0 = 0 or 1 ??

\$t0 = 1 because –1 < +1

sltu \$t0, \$s0, \$s1 # unsigned
\$t0 = 0 or 1???

\$t0 = 0 because +4,294,967,295 > +1

Procedure Calling

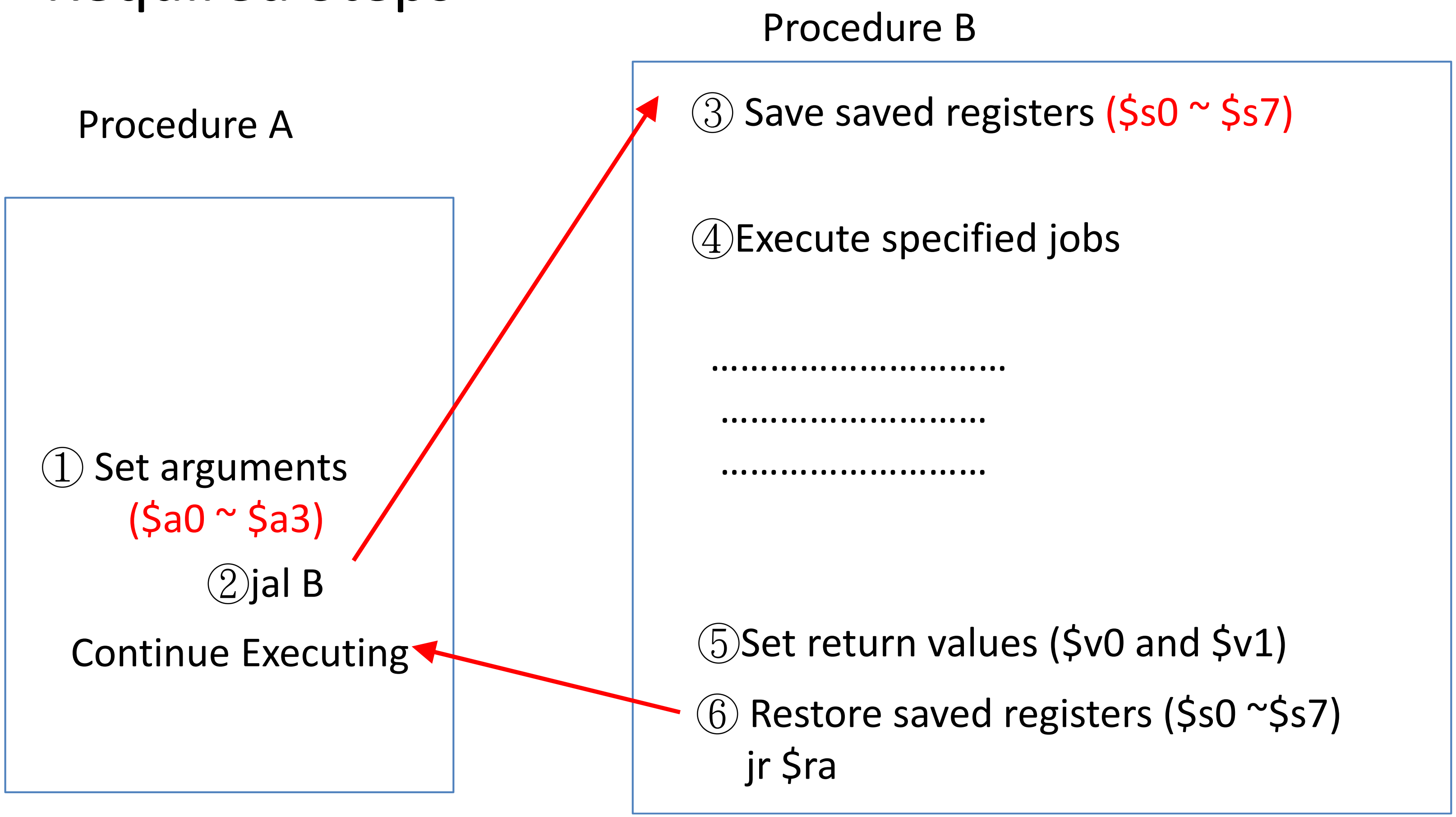
Steps required to execute a procedure

1. Place parameters in registers, where the procedure can access ⇒ Register $a0 to $a3
2. Transfer control to procedure ⇒ Use jal instruction
3. Acquire storage for procedure (save the content of registers that you are going to use)
4. Perform procedure's operations
5. Place results in registers for caller ⇒ (register v0 and v1)
6. Restore saved register and return to place of call ⇒ use jr $ra

Analogy to Spy

Procedure Calling Steps

- Required Steps



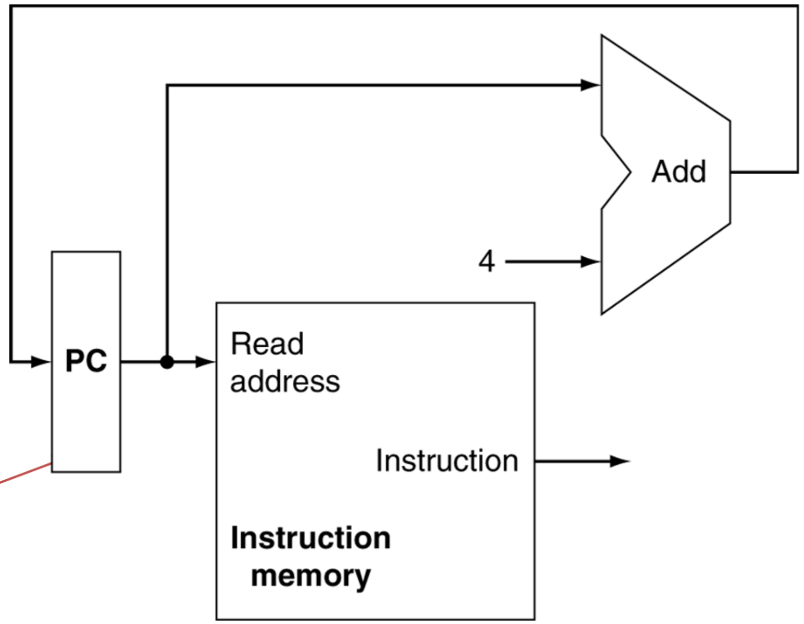
Naming Conventions for procedure calling

- $a0-$a3: four arguments to pass parameters
- $v0-$v1: registers to return values
- $ra : return address
- $t0, $t1….$t9 for temporary values
  - Can be overwritten by callee without saving
- $s0, $s1,….$s7 for saved variable
  - Must be saved/restored by callee

Name	Register number	Usage	Preserved on call?
$zero	0	The constant value 0	n.a.
$v0–$v1	2–3	Values for results and expression evaluation	no
$a0–$a3	4–7	Arguments	no
$t0–$t7	8–15	Temporaries	no
$s0–$s7	16–23	Saved	yes
$t8–$t9	24–25	More temporaries	no
$gp	28	Global pointer	yes
$sp	29	Stack pointer	yes
$fp	30	Frame pointer	yes
$ra	31	Return address	yes

Program Counter

- Program Counter is used to indicate the next instruction to be executed
  - A better name is instruction address register



32-bit register

```
1000 lw ..
1004 add $t0
1008 sll …
100C sll …
…………..
```

Before add (when executing lw)
$pc = 1004

Procedure Call Instructions

- Procedure call: jump and link

  `jal Procedure_Label`

  – Jumps to target address

  – Address of the following instruction are stored in $ra

```
int main()
{   ...
   t1 = fact(8);
   t1 = t1+1;
...
}
int fact(int n)
{
   int i, f = 1;
   for (i = n; i > 1; i--)
   f = f * i;
   return f;
}
```

```
1000 xxx ..
1004 jal fact
1008 … …
100C … …

……………………..

2010 fact:…
2014…………..
2018 ……
```

Before jal
$pc = 1004
$ra = XXX

After jal
$pc = 2010
$ra = 1008

Procedure Call without arguments

```
int main() {
    func1();
    func2();
}
void func1() {}
void func2() {}
```

```
main: li $s0, 7
      jal func1
      jal func2

func1:  addi $t0, $zero, 1   #do something
        jr $ra
func2:  addi $t0, $zero, 2   #do something
        jr $ra
```

jal label

- puts the address of the next instruction into register $ra (return address)
- branches to label

Procedure Call without arguments (preserve s0)

```
int main() {
      func1();
      func2();
}
void func1()
{ subfunc1() }
void func2() {}
```

```
main: li $s0,7
        jal func1
        jal func2
        .....
func1: ....
        li $s0,33
        jr $ra
```

- Both main and func1 uses $s0?
- To avoid conflict
  - Preserve original $s0 in the stack, and then use $s0
  - Restore $s0 when done

Procedure Call without arguments (preserve s0)

```
main: li $s0,7
      jal func1
      jal func2
      .....
func1: ....
      li $s0,33
      jr $ra
```

```
main: li $s0,7
      jal func1
      jal func2
      .....
func1: ....
      sub $sp, $sp, 4
      sw $s0, 0($sp)
      li $s0,33
      lw $s0, 0($sp)
      add $sp, $sp, 4
      jr $ra
```

- What if it uses $s0?

Nested Procedure Call w/o arguments

```
int main() {
      func1();
      func2();
}
void func1()
{ subfunc1() }
```

```
main: li $s0, 7
        jal func1
        jal func2
        .....
func1: ....
        jal subfunc1
        li $s0,33
        jr $ra
```

- What if func1 does a jal? $ra of func1 is overwritten by $ra of subfunc1

Final version: Nested Procedure Call w/o arguments)

```
int main() {
      func1();
      func2(); }
void func1()
{ subfunc1() }
```

Both main & func1 uses $s0, and func1 calls another subfunc1

- save $s0 & $ra => callee behavior

```
main: li $s0,7
        jal func1
        jal func2
        .....
func1: ....
        addi $sp, $sp, -8
        sw $ra 0($sp)
        sw $s0 4($sp)
        jal subfunc1
         li $s0,33
        lw $s0, 4($sp)
        lw $ra, 0($sp)
        addi $sp, $sp, 8
        jr $ra
```

Leaf and Nested Procedures

- Leaf procedure: one that doesn't call other procedures
  - Always a callee
- Nested (nonleaf) procedure: one that calls other procedures => can be both a caller and callee

```
int main() {
      func1();
}
int func1(int arg1, arg2)
{
    int f;
    f = g-h;
    return f;
}
```

func1 procedure is a callee

```
int main() {
      func2();
}
int func2()
{
   int  x ;
    x =func3(arg1,arg2);
   return x;
}
```

Func2 can be both caller and callee

Leaf Procedure Example

- C code:

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

$s0 $t0 $t1

  - Arguments g, …, j in $a0, …, $a3
  - f in $s0 (hence, need to save $s0 on stack)

$a0	g
$a1	h
$a2	i
$a3	j
$s0	f

Three register $s0, $t0, $t1 are saved used in leaf_example because leaf_example need to uses these registers

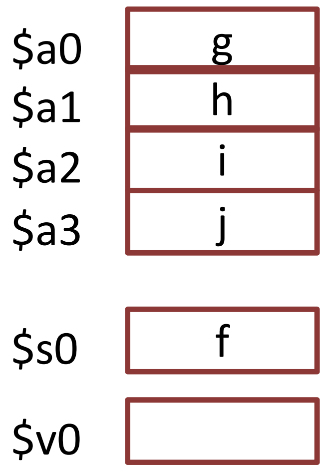
⇨ callee behavior

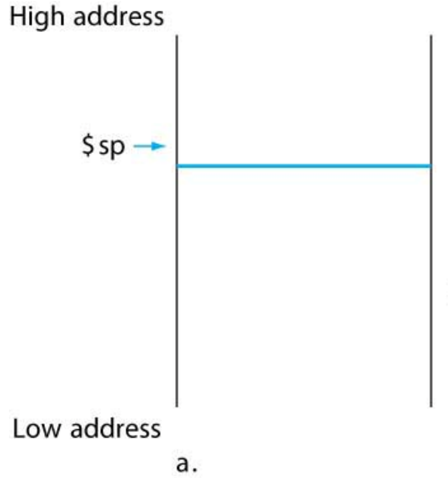
```
int leaf_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

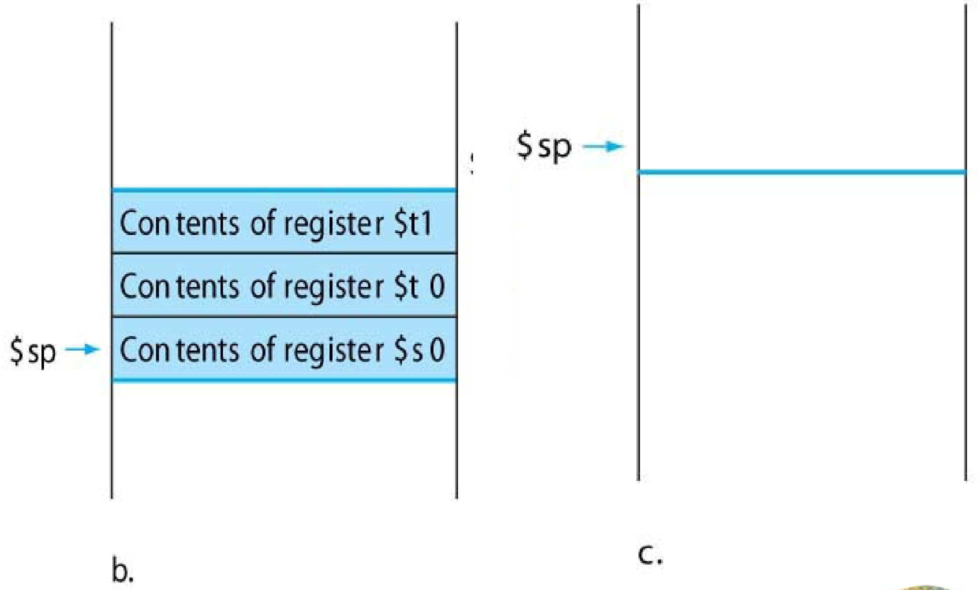
$s0 $t0 $t1

Three local variables

```
leaf_example:
  addi $sp, $sp, -12
  sw   $t1, 8($sp)
  sw   $t0, 4($sp)
  sw   $s0, 0($sp)
  add  $t0, $a0, $a1
  add  $t1, $a2, $a3
  sub  $s0, $t0, $t1
  add  $v0, $s0, $zero
  lw   $s0, 0($sp)
  lw   $t0, 4($sp)
  lw   $t1, 8($sp)
  addi $sp, $sp, 12
  jr   $ra
```







Improved Version

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

$s0 $t0 $t1

Three local variables

$a0	g
$a1	h
$a2	i
$a3	j
$s0	f
$v0	

leaf_example:
`addi $sp, $sp, -12`
~~`sw $t1, 8($sp)`~~
~~`sw $t0, 4($sp)`~~
`sw $s0, 0($sp)`
`add $t0, $a0, $a1`
`add $t1, $a2, $a3`
`sub $s0, $t0, $t1`
`add $v0, $s0, $zero`
`lw $s0, 0($sp)`
~~`lw $t0, 4($sp)`~~
~~`lw $t1, 8($sp)`~~
`addi $sp, $sp, 12`
`jr $ra`

Caller maintains $t0~$t9

Callee maintains $s0~$s7

Therefore, no need to maintain the states of $t0 and $t1

=> 2 sw and 2 lw instructions are reduced

Register State Reservation for leaf procedure

- Maintain register states by saving them before the function executes, and *restoring* them after the function completes.
- Caller handles \$a0-\$a3 and \$t0-\$t9 registers and callee handles \$s0-\$s7 and \$ra.
  - If the callee is a leaf procedure, no need to save \$ra

Save \$s0-\$s7 (callee behavior)

```
int func2(int arg1, arg2)
{
    int f;
    f = g-h;
    return f;
}
```

Restore \$s0-\$s7

No need to save \$ra because callee is a leaf

Register State Reservation for Non-leaf procedure

成功大學

- Caller take care of $a0-$a3 and $t0-$t9 registers and callee take care of $s0-$s7, $ra.
- Nonleaf can be both caller and callee.

```
int func1()
{
  [ ]
 int  x ;
  [ ]
x =func2(…,….);
  [ ]
  return x;
}
```

Save $ra, and $s0-$s7 if necessary (callee behavior)

$a0-$a3 and $t0-$t9 if necessary (caller behavior)

Put parameters in $a0-$a3
Jal callee

restore $t0-$t9
restore $a0-$a3
restore $s0-s7, $ra
restore $ra

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

$a0 | n |
|---|

Register State Reservation

- Caller
  - $a0 ~$a3
  - $t0~$t9
- Callee
  - $ra
  - $s0~$s7

```
fact:
       addi $sp, $sp, -8        # adjust stack for 2 items
       sw   $ra, 4($sp)         # save return address
       sw   $a0, 0($sp)         # save argument
       slti $t0, $a0, 1         # test for n < 1
       beq  $t0, $zero, L1
       addi $v0, $zero, 1       # if so, result is 1
       addi $sp, $sp, 8         #   pop 2 items from stack
       jr   $ra                 #   and return
L1:    addi $a0, $a0, -1        # else decrement n
       jal  fact                # recursive call
       lw   $a0, 0($sp)         # restore original n
       lw   $ra, 4($sp)         #   and return address
       addi $sp, $sp, 8         # pop 2 items from stack
       mul  $v0, $a0, $v0       # multiply to get result
       jr   $ra                 # and return
```

Reserve $ra and $a0

Restore $ra and $a0

What is and what is not preserved across a call

Preserved	Not preserved
Saved registers: $s0–$s7	Temporary registers: $t0–$t9
Stack pointer register: $sp	Argument registers: $a0–$a3
Return address register: $ra	Return value registers: $v0–$v1
Stack above the stack pointer	Stack below the stack pointer

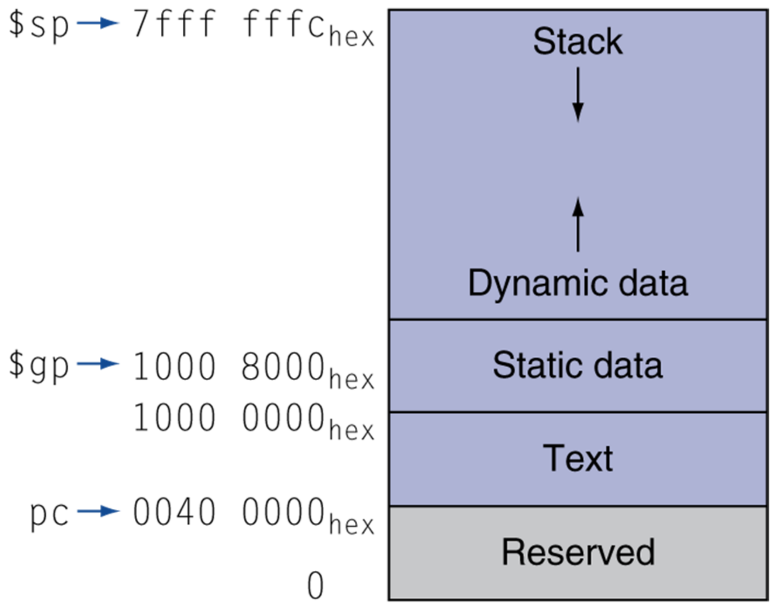
Frame pointer: $fp
Global pointer: $gp

Callee needs to preserve and restore these registers

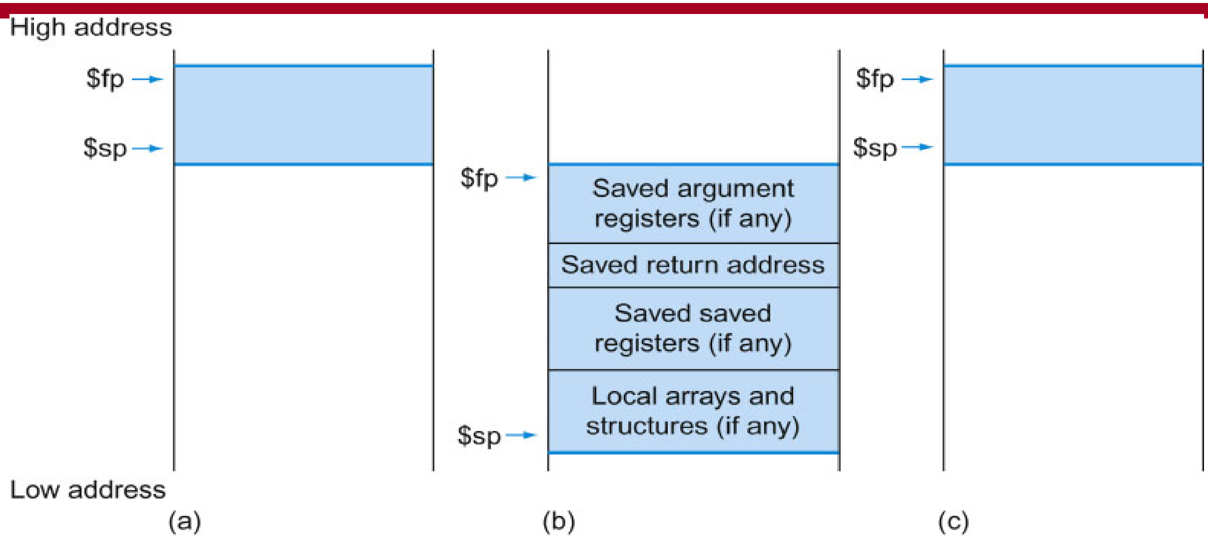
Caller need to preserve and restore these registers

MIPS Memory Layout

- Static Text: program code, start at 00400000_{16}
- Static data: static variables in C, constant arrays and strings
  - Start at $1000\ 0000_{16}$
  - $gp (global pointer) initialized to 10008000_{16}
    - allowing $\pm$offsets into this segment
- Dynamic data: heap,
  - E.g., malloc in C, new in Java
  - Grow up toward stack
- Stack: automatic storage
  - $sp initialized to $7ffffffc_{hex}$



Stack allocation



Before stack allocation

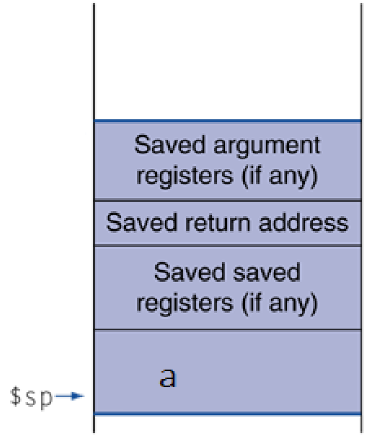
During stack allocation

After stack allocation

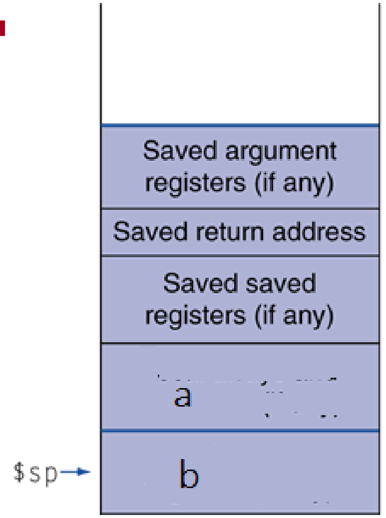
- Stack are allocation during procedure call
- Stack pointer ($sp) may change during program execution => use stable frame point to access data (see next slides)

Local Data on the Stack

```
        Func test() {
1  ───►  int a;
         .... Stmts..
2  ───►  int b;
         .....Stmts...
        }
```



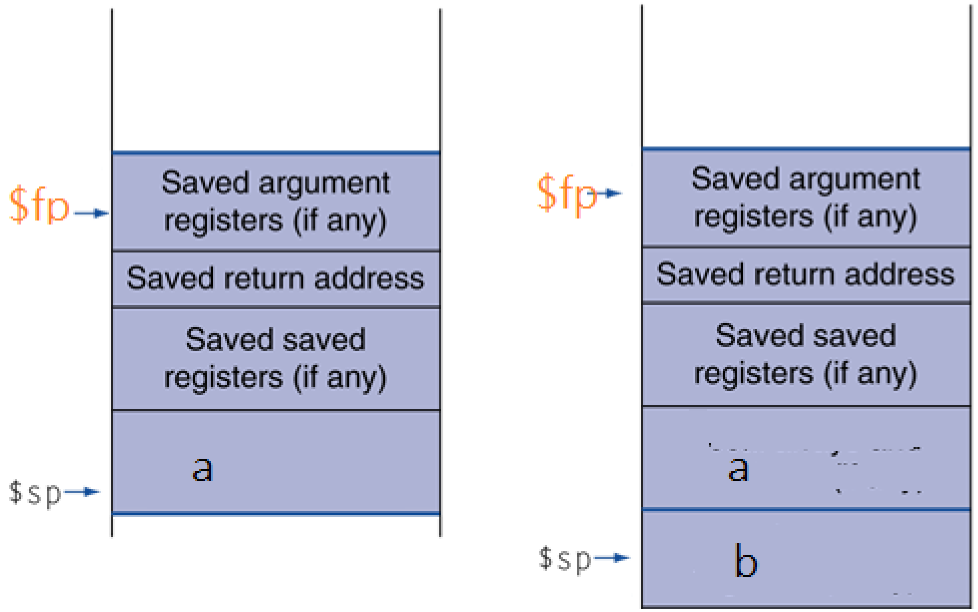
At 1



At 2

- local data are also preserved in a procedure
  - e.g., C automatic variables
- Therefore, $sp value may change in procedure => local variable have different offsets
  - e.g. a is $sp in case 1, and $sp+4 in case 2
  - hard to use $sp to access local data
  - Define a new pointer => $fp

Frame pointer




```
Func test() {
  int a;
  ....  //statements..
   int b;
   ...../statements
}
```

- Frame pointer ($fp) points to the first word of the frame of a procedure (activation record)
- Frame pointer : a stable base register for local memory-reference

Summary: Register Conventions

- Caller handle of $a0-$a3 and $t0~$t9, $t8, $t9
- Callee handle $ra and $s0~$s7, $gp, $sp, $fp are preserved on a procedure call

Name	Register number	Usage	Preserved on call?
`$zero`	0	The constant value 0	n.a.
`$v0–$v1`	2–3	Values for results and expression evaluation	no
`$a0–$a3`	4–7	Arguments	no
`$t0–$t7`	8–15	Temporaries	no
`$s0–$s7`	16–23	Saved	yes
`$t8–$t9`	24–25	More temporaries	no
`$gp`	28	Global pointer	yes
`$sp`	29	Stack pointer	yes
`$fp`	30	Frame pointer	yes
`$ra`	31	Return address	yes

Syscall

#Print a integer

```
.data
    age: .word 32
.text
  #print an integer to the screen
  li $v0, 1
  lw $a0, age
  syscall
```

#Print a character

```
.data
   myCharacter: .byte 'a'
.text
  li $v0, 4
  la $a0, myCharacter
  syscall
```

```
.data
   myMessage: .asciiz "Hello world Ing Chao Lin\n"

.text
  li $v0, 4
  la $a0, myMessage
  syscall
```

```
.data
.text
li $v0, 10
syscall
```

Program stops

Backup slides