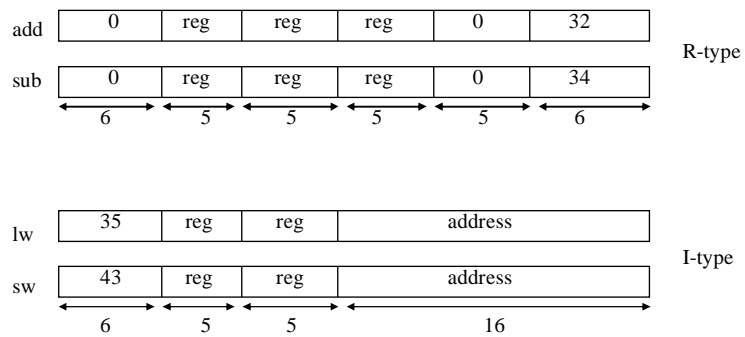

Chapter 2.7 ~ 9

Summary of the last class

Representing instructions in the computer



2.7 Supporting Procedure Calls

```
.....
leaf_example (1, 2, 3, 4);
.....
```

1. Place parameters in some place
2. Transfer control to the procedure
3. Perform the task
4. Place the result in some place
5. Return control to the original point

```
.....
int leaf_example (int g, int h, int i, int j)
{
    int f;

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

Policy of Register Use Conventions

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

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

1. Place parameters in some place
2. Transfer control to the procedure
3. Perform the task
4. Place the result in some place
5. Return control to the original point

Supporting Procedure Calls

```
.....  
leaf_example (1, 2, 3, 4);  
.....  
.....  
int leaf_example (int g, int h, int i, int j)  
{  
    int f;  
  
    f = (g+h) - (i+j);  
    return f;  
}
```

1. ...
2. Transfer control to the procedure
jal leaf_example

jump-and-link instruction
saves PC+4 to \$ra
and jump to leaf_example

leaf_example:
3. ...
4. ...
5. Return control to the original point
jr \$ra

\$ra points the next instruction
of the calling instruction

Supporting Procedure Calls

```
.....  
leaf_example (1, 2, 3, 4);  
.....  
.....  
int leaf_example (int g, int h, int i, int j)  
{  
    int f;  
  
    f = (g+h) - (i+j);  
    return f;  
}
```

1. Place parameters
\$a0=1, \$a1=2, \$a2=3, \$a3=4
2. ...

leaf_example:
3. ...
4. Place the results
\$v0, \$v1=...

5. ...

Supporting Procedure Calls

```

.....
leaf_example (1, 2, 3, 4);
.....

.....
int leaf_example (int g, int h, int i, int j)
{
    int f;

    f = (g+h) - (i+j);
    return f;
}

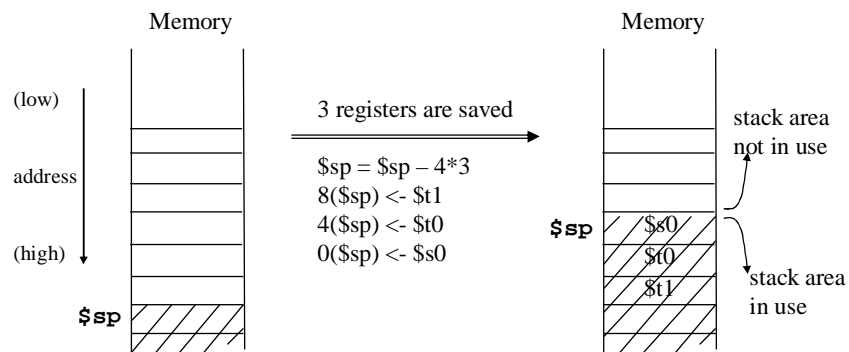
```

\$s0 = 43;
 1. ...
 2. ...
 \$t3 = \$s0 + \$v0 (result register)
 # but is \$s0 still "43" ???

leaf_example:
 # may change some registers
 # (\$s0, \$t0, \$t1)
 # save them into stack (push)
 3. Perform the task
 # restore from stack (pop)
 4. ...
 5. ...

Stack

- Last-in first-out
- "Push" for inputting, and "pop" for retrieving
- Stack pointer (\$sp) points to the top of the stack



Supporting Procedure Calls

.....
leaf_example (1, 2, 3, 4);
.....

1. ...
2. ...

* Convention is to save
\$s0-\$s7 but not \$t0-\$t9
(Save: \$s0-7, \$sp, \$ra
No: \$t0-9, \$a0-3, \$v0-1)

.....
int leaf_example (int g, int h, int i, int j)
{
 int f;

 f = (g+h) - (i+j);
 return f;
}

leaf_example:

save them into stack (push)
sub \$sp, \$sp, 12
sw \$t1, 8(\$sp)
sw \$t0, 4(\$sp)
sw \$s0, 0(\$sp)

3. Perform the task
restore from stack (pop)
lw \$s0, 0(\$sp)
lw \$t0, 4(\$sp)
lw \$t1, 8(\$sp)
add \$sp, \$sp, 12

4. ...
5. ...

Stack for Local Variables (automatic<-> static)

.....
leaf_example (1, 2, 3, 4);
.....

1. ...
2. ...

.....
int leaf_example (int g, int h, int i, int j)
{
 int f;

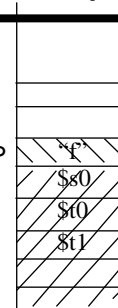
 f = (g+h) - (i+j);
 return f;
}

leaf_example:

save them into stack (push)
sub \$sp, \$sp, 12
sw \$t1, 8(\$sp)
sw \$t0, 4(\$sp)
sw \$s0, 0(\$sp)
reserve a location for local variable
sub \$sp, \$sp, 4

3. Local variable "f" is replaced by (\$sp)
4. ...
5. ...

\$sp



Summary: Supporting Procedure Calls

```

.....
leaf_example (1, 2, 3, 4);
.....

```

1. Place parameters
\$a0=1, \$a1=2, \$a2=3, \$a3=4
2. Transfer control to the procedure
jal leaf_example

```

.....
leaf_example:
int leaf_example (int g, int h, int i, int j)
{
  int f;

  f = (g+h) - (i+j);
  return f;
}

```

3. Perform the task

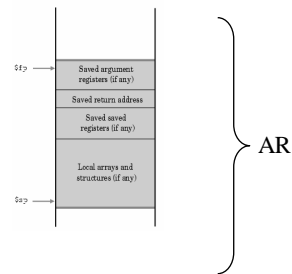

```

sub $sp, $sp, 12 # update $sp
sw $t1, 8($sp) # push
sw $t0, 4($sp)
sw $s0, 0($sp)
sub $sp, $sp, 4 # local variable
add $sp, $sp, 4 # local variable
lw $s0, 0($sp) # pop
lw $t0, 4($sp)
lw $t1, 8($sp)
add $sp, $sp, 12 # update $sp

```
4. Place the results
\$v0, \$v1=...
5. Return control to the original point
jr \$ra

More on Stack

- Stack for registers and local variables
- Procedure frame = activation record
 - Segment of stack which has a procedure's saved registers and local variables
- Frame pointer (\$fp)
 - Is used to reference a local variable stored in stack
 - Why \$sp does not do that ?
- What is bad with recursive procedures ?



- When does stack overflow occur ?
← too many nest calls
- When does stack underflow occur ?
← error

```

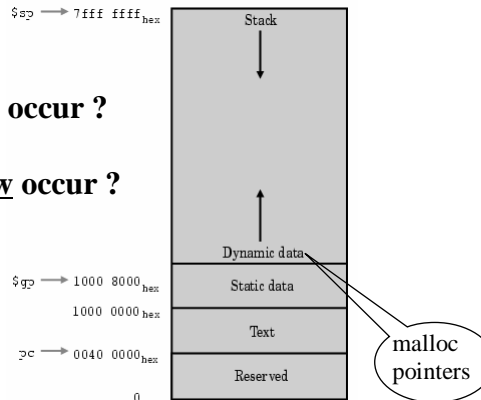
{...
...
{int c;
...
}
}

```

More on Stack

• When does stack overflow occur ?

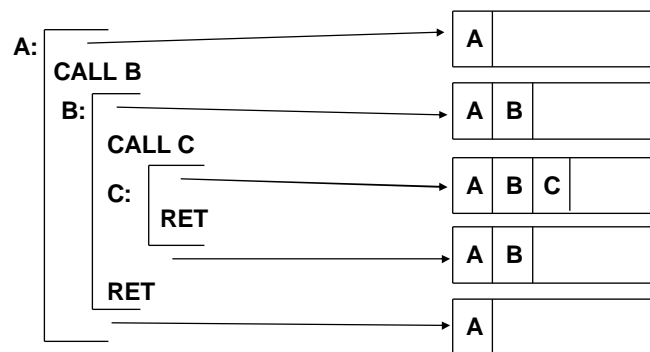
• When does stack underflow occur ?



Dynamic data area = heap area

- Linked list created by malloc() and freed y free()
- Many problems such as memory leak and dangling pointer

More on Stack



Some machines provide a memory stack as part of the architecture (e.g., VAX)

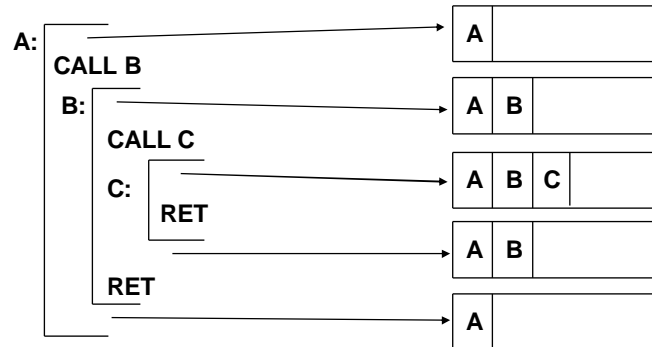
Sometimes stacks are implemented via software convention (e.g., MIPS)

Is this possible?

A	A	A
---	---	---

A	B	A
---	---	---

More on Stack



- Stack for registers and local variables is called **“activation record”**
- What is bad with **recursive procedures** ?

2.8 Communication with People: Byte Data & Constants

- **Character**
 - Is a byte quantity (00~FF or 0~255)
 - ASCII (American Standard Code for Information Interchange)
 - Page 91, Fig. 2.21

32: space	33: !	34: “	35: # ...	
48: 0	49: 1 ...	64: @	65: A	66: B ...
97: a	98: b ...	127: DEL		
 - What others ?
 - 0~31: control characters
 - 0: ^@ (NUL) 1: ^A 2: ^B ... 7: ^G (BEL)
 - 8: ^H (BS) ... 13: ^M (CR)... 31: ^- (US, cursor up)
 - 128~255: graphic characters
- **String**
 - A sequence of characters
 - In “C” language, it is “null-terminated”: “Cab” = 67 97 98 0

Byte Data & Constants

- How to load/store characters from/to memory? (not “integers”)
- Example:
 - Copy of the four characters from [160] to [200]
 - Assume \$s0=160, \$s1=200
 - lw \$t0, 0(\$s0) & sw \$t0,0(\$s1): OK

– What if it is “Caba” = 67 97 98 97 0 ?

– lw \$t0, 0(\$s0) & sw \$t0, 0(\$s1)

– lw \$t1, 4(\$s0) & sw \$t1, 4(\$s1)

=> we also copied 3 un-wanted bytes in 165-167

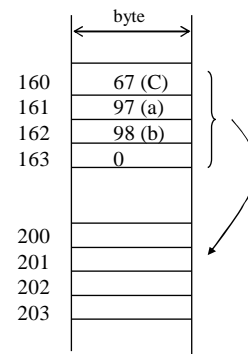
=> we want load/store inst. in byte quantity

=> lb & sb

=> the second inst must be

lb \$t1, 4(\$s0) & sb \$t1, 4(\$s1)

Must be word boundary?



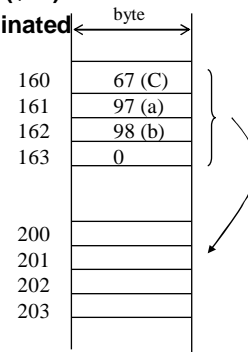
Byte Data & Constants

- Exercise : strcpy(a[], b[])

– Assume a[] starts at 160 (\$s0), b[] starts at 200 (\$s1)

– We don't know the string size, but it is null-terminated

– \$zero register can be used



MIPS data transfer instructions

<i>Instruction</i>	<i>Comment</i>	
SW R3, 500(R4)	Store word	Sign-extended (affect the entire word)
SH R3, 502(R2)	Store half	
SB R2, 41(R3)	Store byte	
LW R1, 30(R2)	Load word	Not sign-extended (affect the entire word) (i.e., fill with 0's in upper 2 or 3 bytes)
LH R1, 40(R3)	Load halfword	
LHU R1, 40(R3)	Load halfword unsigned	
LB R1, 40(R3)	Load byte	
LBU R1, 40(R3)	Load byte unsigned	
LUI R1, 40	Load Upper Immediate (16 bits shifted left by 16) (later in this lecture)	

2.9 MIPS Addressing: Constants & Addressing

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

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

- Solutions
 - put 'typical constants' in memory and load them ???
 - create hard-wired registers (like \$zero) for constants ???

- MIPS Instructions:

```
addi $s0, $s0, 4
slti $s1, $s2, 10
andi $t1, $t2, 6
ori $t2, $t2, 4
```

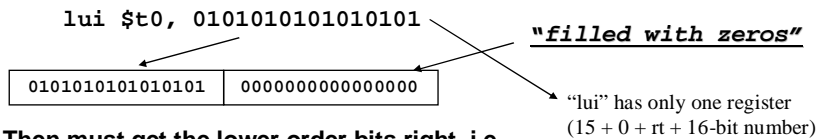
- R-type or I-type?
- What's the range of the number?
- addi & addui
- with "addui", the maximum is $2^{16}-1$
=> what if we want a larger constant ?

- "i" means immediate data

How about larger constants?

1,431,695,600 = (0101 0101 0101 0101 1111 0000 1111 0000)

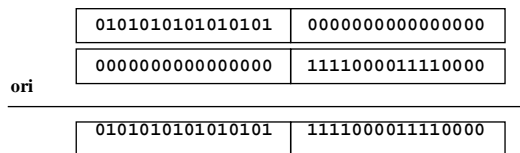
- We'd like to be able to load a 32 bit constant into a register
- Must use two instructions, new "load upper immediate" instruction



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

ori \$t0, \$t0, 1111000011110000

- “addui” can also be used
- Why “addi” can't be used?
=> negative !!!



Addresses in Branches and Jumps

- Instructions:

bne \$t4,\$t5,Label	Next instruction is at Label if \$t4 != \$t5
beq \$t4,\$t5,Label	Next instruction is at Label if \$t4 = \$t5
j Label	Next instruction is at Label

- Formats:

I	op	rs	rt	16 bit address
J	op	26 bit address		

- How to generate a 32-bit address with a 16 or 26-bit number?

- How do we handle this with load and store instructions?
=> memory address = \$rs + 16-bit offset = 32-bit quantity: OK
- How about in beq/bne?
- How about in j?

Addresses in Branches

- Instructions:

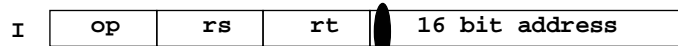
`bne $t4, $t5, Label`

Next instruction is at Label if $\$t4 \neq \$t5$

`beq $t4, $t5, Label`

Next instruction is at Label if $\$t4 = \$t5$

- Formats:



- Next PC = current PC + Label : "PC-relative addressing"

- PC = program counter

- most branches are local (principle of locality)

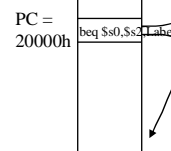
- Branching range is ???

- When "current PC" = 20000h

- The jumping address is 20000~2ffff ??? (Sign bit)

- The jumping address is $20000 - 7fff \sim 2000 + 7fff = 18001 \sim 27fff$

- $\Rightarrow 20000 - 7fff * 4 \sim 20000 + 7fff * 4 = 0 \sim 40000 (-4)$



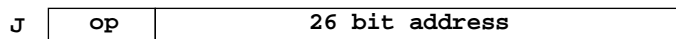
Addresses in Jumps

- Instructions:

`j Label`

Next instruction is at Label

- Formats:



- Next PC = First 4-bit of current PC + Label (26-bit)

- just use high order bits of PC

- address boundaries of 256 MB ???

$4 + 26 = 30 \neq 32$???

In effect, 28-bit.
Why ???

MIPS Addressing Modes

1. Immediate addressing



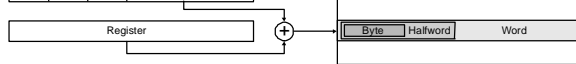
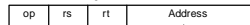
`addi $s0, $s1, 4`

2. Register addressing



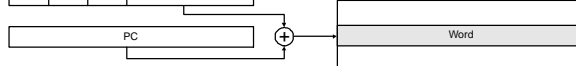
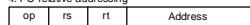
`add $s0, $s1, $s2`

3. Base addressing



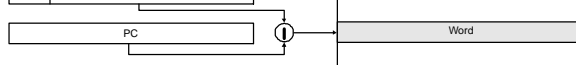
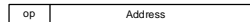
`lw $s0, 32($s1)`

4. PC-relative addressing



`bne $s0, $s1, Label`

5. Pseudodirect addressing



`j Label`

MIPS Addressing Modes

- Decoding or reverse engineering (page 153, Fig.3.18)
- Exercise
 - Which instruction is this?
 - 1000 1110 0101 0001 0000 0000 0110 0100
 - => 6-bit + (5-bit + 5-bit + (5-bit + 5-bit + 6-bit))
 - => 100011 : 35 = lw => 6+5+5+16
 - => 100011 10010 10001 0000000001100100
 - => lw \$s2 \$s1 100
 - => lw \$s1, 100(\$s2)
 - 0000 0010 0101 0011 1000 1000 0010 1010
 - => 000000 : 0 = need to look at funct: 101010:42 = slt => 6+5+5+5+5+6
 - => 000000 10010 10011 10001 00000 101010
 - => 0 \$s2 \$s3 \$s1 shamt slt
 - => slt \$s1, \$s2, \$s3
- But
 - Is memory restricted to 32-bit addressable range or 4GB ?
 - What if we have 64-bit memory space ?

To summarize:

MIPS operands

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

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add \$s1, \$s2, \$s3	\$s1 = \$s2 + \$s3	Three operands; data in registers
	subtract	sub \$s1, \$s2, \$s3	\$s1 = \$s2 - \$s3	Three operands; data in registers
	add immediate	addi \$s1, \$s2, 100	\$s1 = \$s2 + 100	Used to add constants
Data transfer	load word	lw \$s1, 100(\$s2)	\$s1 = Memory[\$s2 + 100]	Word from memory to register
	store word	sw \$s1, 100(\$s2)	Memory[\$s2 + 100] = \$s1	Word from register to memory
	load byte	lb \$s1, 100(\$s2)	\$s1 = Memory[\$s2 + 100]	Byte from memory to register
	store byte	sb \$s1, 100(\$s2)	Memory[\$s2 + 100] = \$s1	Byte from register to memory
	load upper immediate	lui \$s1, 100	\$s1 = 100 * 2 ¹⁶	Loads constant in upper 16 bits
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 less than immediate	slti \$s1, \$s2, 100	If (\$s2 < 100) \$s1 = 1; else \$s1 = 0	Compare less than constant
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