CO101
Principle of Computer Organization

Lecture 03: Instruction Set Architecture

Liang Yanyan

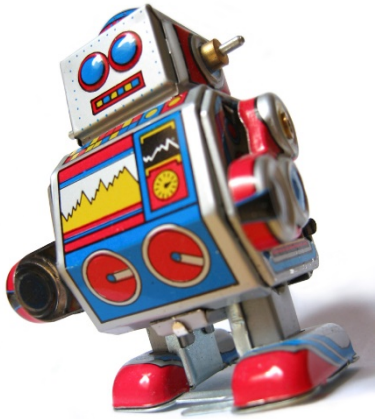
澳門科技大學
Macau of University of Science and Technology

Review: The Instruction Set Architecture (ISA)

software

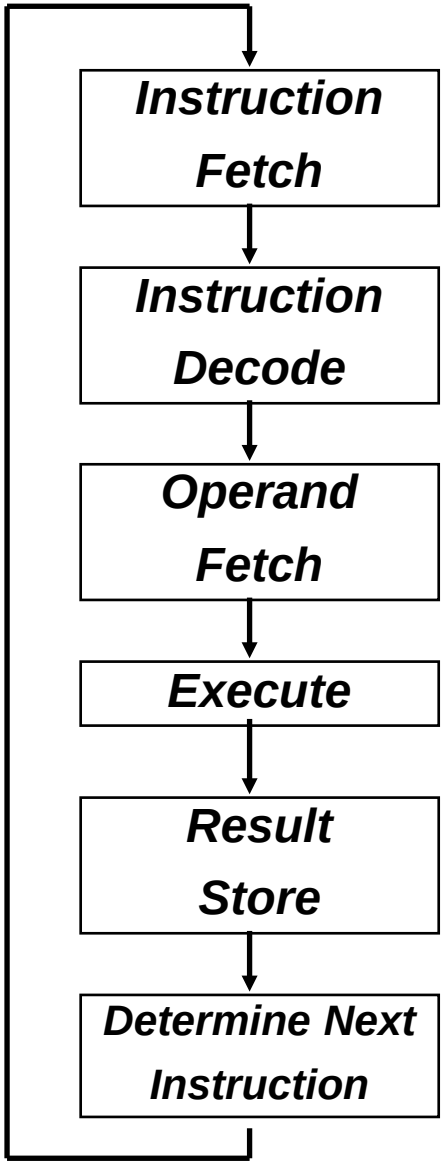
instruction set

hardware



The interface description separating
the software and hardware

Review: Execution Cycle



Obtain instruction from program storage

Determine required actions and instruction size

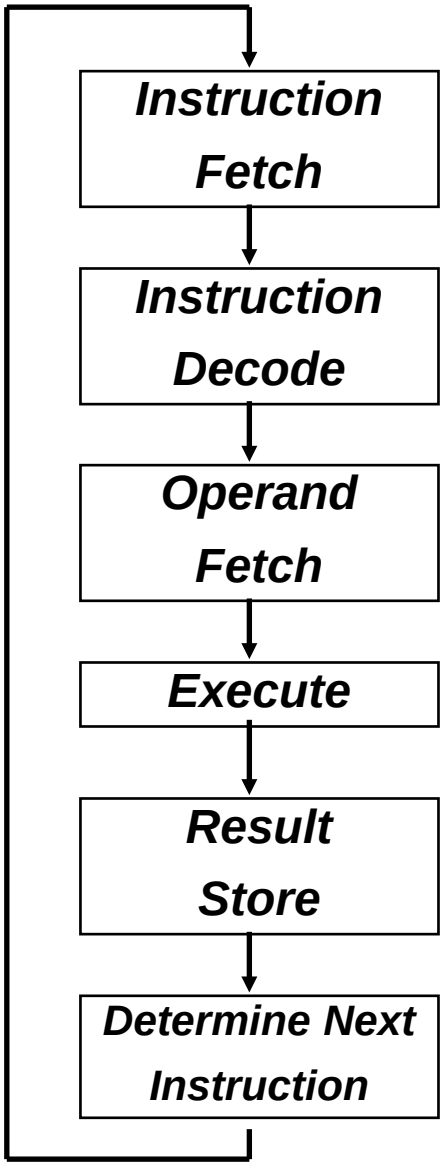
Locate and obtain operand data

Compute result value or status

Deposit results in storage for later use

Determine successor instruction

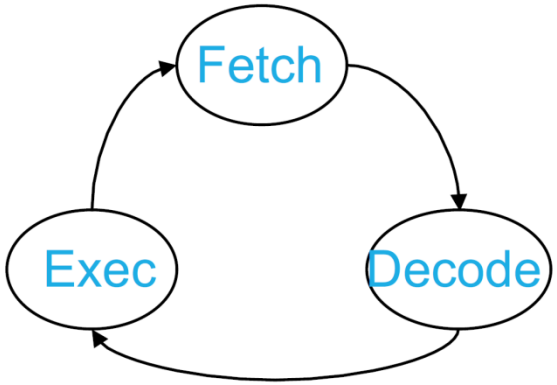
ISA: What must be specified?

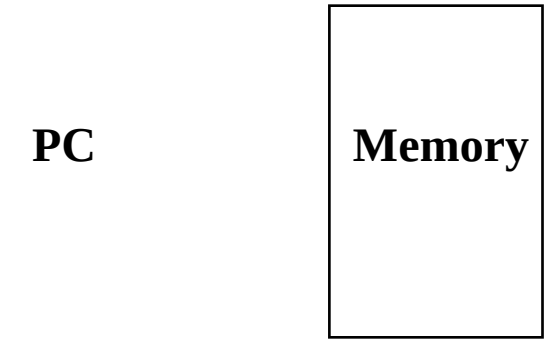


- Where are instructions stored?
- Instruction Format or Encoding
  - how is it encoded?
- Location of operands and result
  - where other than memory?
  - how are memory operands located?
- Data type and Size
- Operations
  - what operations are supported?
- How to determine successor instruction?
  - normal instruction, jumps, branches

Review: Processor Organization

- Fetch & Execute Cycle
  - Instructions are fetched from memory one by one.
  - Bits in the instruction "control" the subsequent actions.
  - Fetch the “next” instruction and continue.





PC is a special register called Program Counter. It stores the memory address of current instruction being executed. Processor fetch instruction addressed by PC to execute.

Review: Processor Organization

- Control needs to have circuitry to
  - Decide which is the next instruction and input it from memory.
  - Decode the instruction.
  - Issue signals that control the way information flows between datapath components.
  - Control what operations the datapath's functional units perform.
- Datapath needs to have circuitry to
  - Execute instructions – functional units (e.g., adder) and storage locations (e.g., register file).
  - Interconnect the functional units so that the instructions can be executed as required.
  - Load data from and store data to memory.

Stored Program Concept

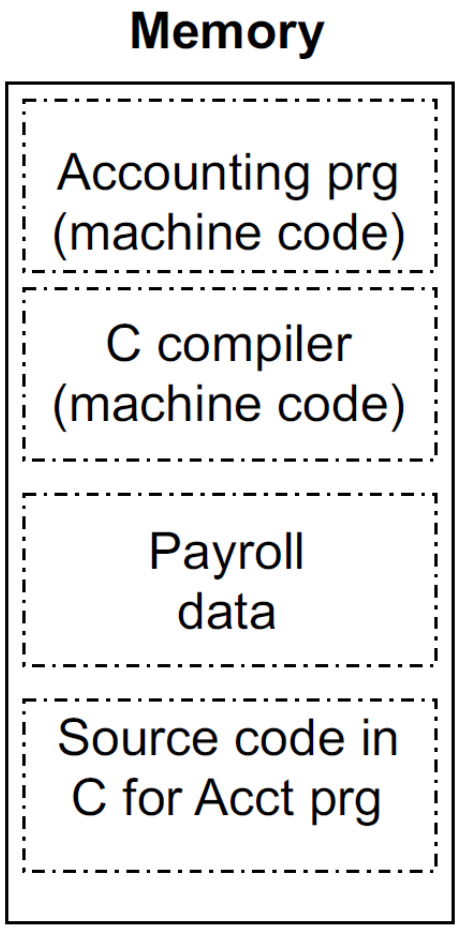
- Today's computers are built on two key principles:
  - Instructions are represented as numbers.
  - Programs are stored in memory to be read or written, just like numbers.

- These principles lead to the *stored-program* concept; its invention let the computing genie out of its bottle. (BTW, "let genie out of its bottle" is a famous idiom)

- Stored-program concept
  - Programs can be shipped as files of binary numbers – binary compatibility.
  - Computers can inherit ready-made software provided they are compatible with an existing ISA---leads industry to align around a small number of ISA.

Two Key Principles of Machine Design

- Machine instructions are bits.
  - Instructions are represented as numbers and, as such, are indistinguishable from data.
- Programs are stored in memory to be read or written just like data.
- Memory can contain the source code for an editor program, the corresponding compiled machine code, the text that the compiled program is using, and even the compiler that generated the machine code. (All above are stored as bits) .



Assembly Language

- Language of the machine.
- Instructions in assembly language are more primitive than instructions in higher level languages.
  - each instruction controls the machine to finish one operation.
  - robot example: forward, backward, etc.
  - computer example: add, subtract, multiply, divide, etc.
- Very restrictive.
  - e.g., MIPS Arithmetic Instructions.
- This course focuses on MIPS assembly language.
  - Similar to other ISAs developed since the 1980's.
  - Used by Broadcom, Cisco, NEC, Nintendo, Sony, …

Assembly Language Instructions

- The language of the machine
  - Want an ISA that makes it *easy* to build the hardware and the compiler while maximizing performance and minimizing cost.

Design goals: maximize performance, minimize cost, reduce design time (time--to--market), minimize memory space (embedded systems), minimize power consumption (mobile systems)

RISC -- Reduced Instruction Set Computer

- RISC philosophy
  - fixed instruction lengths
  - load--store instruction sets
  - limited number of addressing modes
  - limited number of operations
- MIPS, Sun SPARC, HP PA--RISC, IBM PowerPC …
- Instruction sets are measured by how well compilers use them as opposed to how well assembly language programmers use them.

- CISC (C for complex), e.g., Intel x86

MIPS (RISC) Design Principles

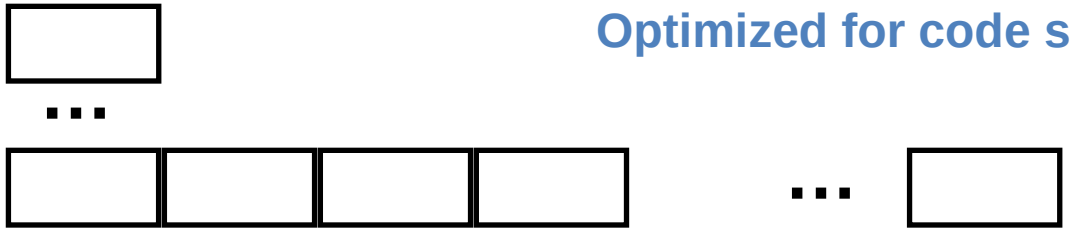
- Simplicity favors regularity
  - fixed size instructions
  - small number of instruction formats
  - opcode always the first 6 bits
- Smaller is faster
  - limited instruction set
  - limited number of registers in register file
  - limited number of addressing modes
- Make the common case fast
  - arithmetic operands from the register file (load--store machine)
  - allow instructions to contain immediate operands
- Good design demands good compromises
  - three instruction formats

RISC vs CISC

- RISC (Reduced instruction set computer)
  - e.g. MIPS, ARM, SPARC by Sun Microsystems
  - provide simplified instructions
  - goal is to reduce the execution time of each instruction
  - drawback is longer code size
- CISC (Complex instruction set computer)
  - e.g. x86
  - provide more powerful operations
  - goal is to reduce number of instructions executed → reduce code size
  - makes assembly language easy
  - danger is longer to execute one instruction
- Ease of compilation vs hardware complexity
  - Complex hardware makes compilation easy, vice versa.

Generic Examples of Instruction Format Widths

Variable:

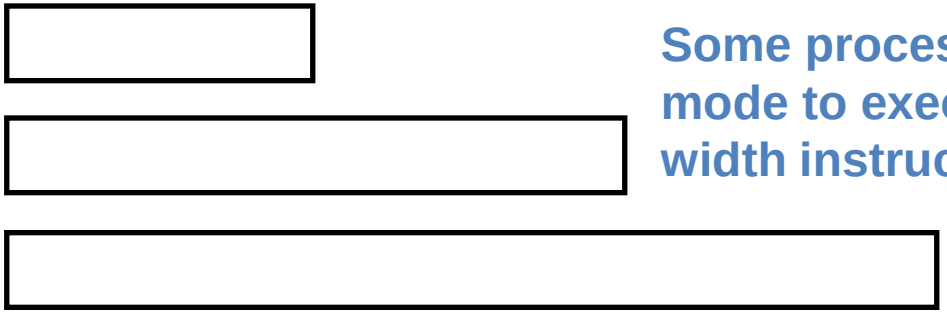


Optimized for code size

Fixed:

Optimized for performance (speed)

Hybrid:

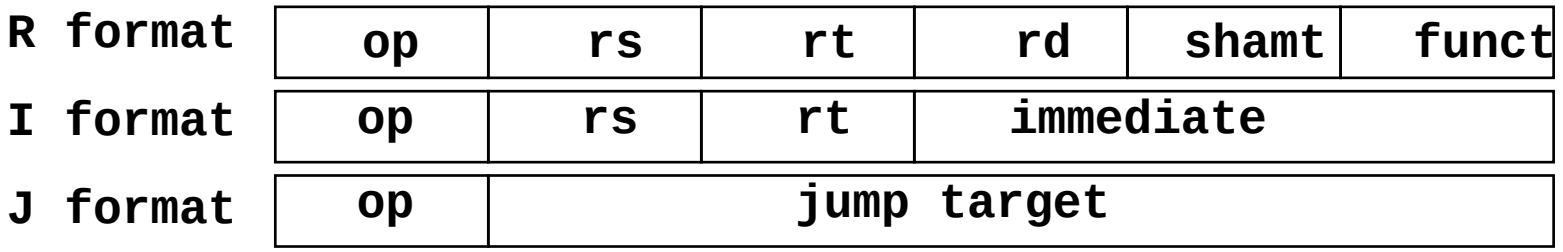


Some processors provide optional mode to execute subset of 16-bit width instructions.

Arithmetic, Memory, Branch instructions

- Instruction Categories
  - Computational
  - Load/Store
  - Jump and Branch
  - Floating Point
  - coprocessor
  - Memory Management
  - Special

- Divide MIPS instructions into 3 categories
- Machine formats:
  - 3 formats, all 32 bits wide
  - Very structured, rely on compiler to achieve performance



MIPS Instruction Classes Distribution

- Frequency of MIPS instruction classes for SPEC2006

Instruction Class	Frequency	
	Integer	Ft. Pt.
Arithmetic	16%	48%
Data transfer	35%	36%
Logical	12%	4%
Cond. Branch	34%	8%
Jump	2%	0%

MIPS Arithmetic Instructions

- MIPS assembly language arithmetic statement

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

- Each arithmetic instruction performs one operation
- Each specifies exactly three operands that are all contained in the datapath's register file ($t0,$s1,$s2)

destination = source1 op source2

- Instruction Format (R format)

MIPS Logical Instructions

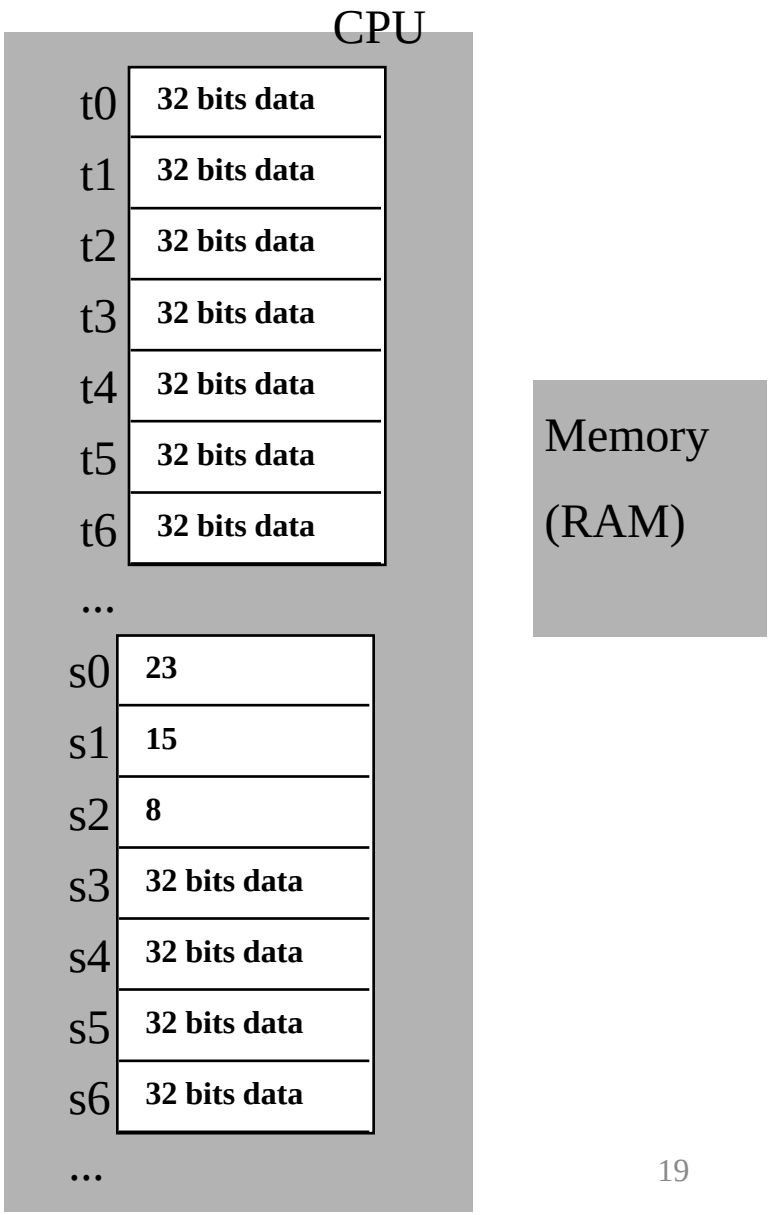
- There are a number of bit--wise logical operations in the MIPS ISA

```
and $t0, $t1, $t2 #$t0 = $t1 & $t2
or  $t0, $t1, $t2 #$t0 = $t1 | $t2
nor $t0, $t1, $t2 #$t0 = not($t1 | $t2)
```

- Instruction Format (R format)

MIPS Arithmetic Instructions

- Perform arithmetic calculations.
  - All instructions have 3 operands which must be registers.
  - Operand order is fixed (destination first).
- There 32 general purpose registers in CPU, each one has a name. E.g. s0, s1, s2, etc.
- Each register can store a 32-bit data/value.
- `add $s0, $s1, $s2`
  - Meaning: add the value stored in register s1 with the value stored in register s2, and store the sum into register s0.
  - E.g. if s1 stores a data 15, s2 stores a data 8, then the sum 23 will be stored into register s0 when this instruction is executed.
- `sub $s0, $s1, $s2 ??`



MIPS Arithmetic Instructions

- Example:

  C code: `A = B + C`

  MIPS code:
  ```
  add $s0, $s1, $s2
  ```

  (associated with variables by compiler)

- Of course this complicates some things... code size

  C code: `A = B + C + D;`

  MIPS code:
  ```
  add $t0, $s1, $s2
  add $s0, $t0, $s3
  ```

Registers vs. Memory

- Why register?
  - registers are faster than memory.
  - using registers can reduce memory transfer.

  Example:

```
a = b + c;
d = a + b;
```

  If there is no registers, how many memory reads/writes?

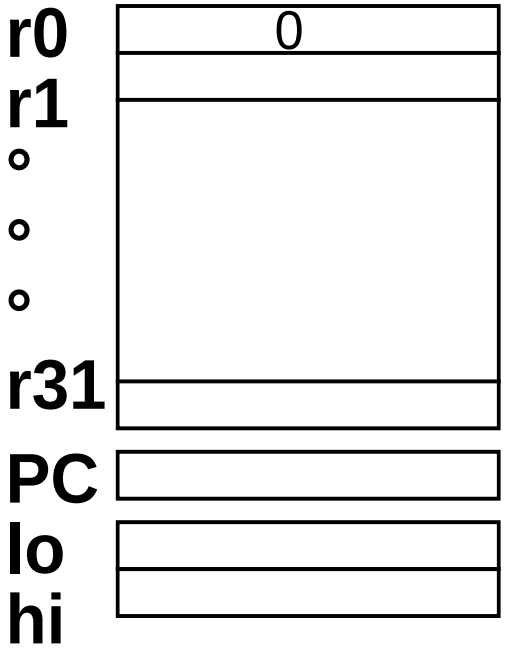
- What about programs with many variables?
  - Reuse registers.

MIPS Register File

- A small piece of memory inside the processor to store values.
- Advantages of registers
  - registers are faster than memory But register files with more locations are slower (e.g., a 64 word file could be as much as 50% slower than a 32 word file).
  - registers are easier for a compiler to use
    - e.g., (A*B) – (C*D) – (E*F) can do multiplies in any order vs. stack
  - Can hold variables so that
    - memory traffic is reduced, so program is speed up (since registers are faster than memory).
    - code density improves (since register named with fewer bits than memory location).

MIPS Register File

- Programmable storage
  - 2^32 bytes of memory
  - 31 x 32-bit General Purpose Registers (R0 = 0)
  - 32 x 32-bit Floating Point Registers
  - PC - program counter
  - lo hi - multiplier output registers



Aside: MIPS Register Convention

0	zero	constant 0
1	at	reserved for assembler
2	v0	expression evaluation &
3	v1	function results
4	a0	arguments (proc. call)
5	a1	
6	a2	
7	a3	
8	t0	temporary
. . .		(callee can clobber)
15	t7	
16	s0	preserved, callee
		saves and restore
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 (HW)

MIPS Arithmetic Instructions

- MIPS assembly language arithmetic statement

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

- Each arithmetic instruction performs one operation
- Each specifies exactly three operands that are all contained in the datapath's register file ($t0,$s1,$s2)

destination = source1 op source2

- Instruction Format (R format)

MIPS Logical Instructions

- There are a number of bit--wise logical operations in the MIPS ISA

```
and $t0, $t1, $t2 #$t0 = $t1 & $t2
or  $t0, $t1, $t2 #$t0 = $t1 | $t2
nor $t0, $t1, $t2 #$t0 = not($t1 | $t2)
```

- Instruction Format (R format)

MIPS Shift Instructions

- Need operations to pack and unpack 8--bit characters into 32--bit words.
- Shifts move all the bits in a word left or right.

```
sll $t2, $s0, 8 #$t2 = $s0 << 8 bits
srl $t2, $s0, 8 #$t2 = $s0 >> 8 bits
```

- Such shifts are called logical because they fill with zeros.

MIPS Immediate Instructions

- Small constants are used quite frequently (50% of operands) Solutions? Why not?
  - put 'typical constants' in memory and load them.

    e.g. `count: .word 35`

    this assembly instruction defines a constant value 35 and this data is stored in memory. The variable "count" is the memory address of the data 35.
  - create hard-wired registers like $zero, which is special register always store a zero.

- MIPS Arithmetic Instructions with constants:

```
e.g.  addi $s1, $zero, 5     (A = 5)
      addi $s2, $s2, 4       (A = A + 4)

      slti $s5, $s5, 10
      andi $s2, $s2, 6       (A = A & 6)
      ori  $s2, $s2, 4       (A = A | 4)
```

MIPS Memory Access Instructions

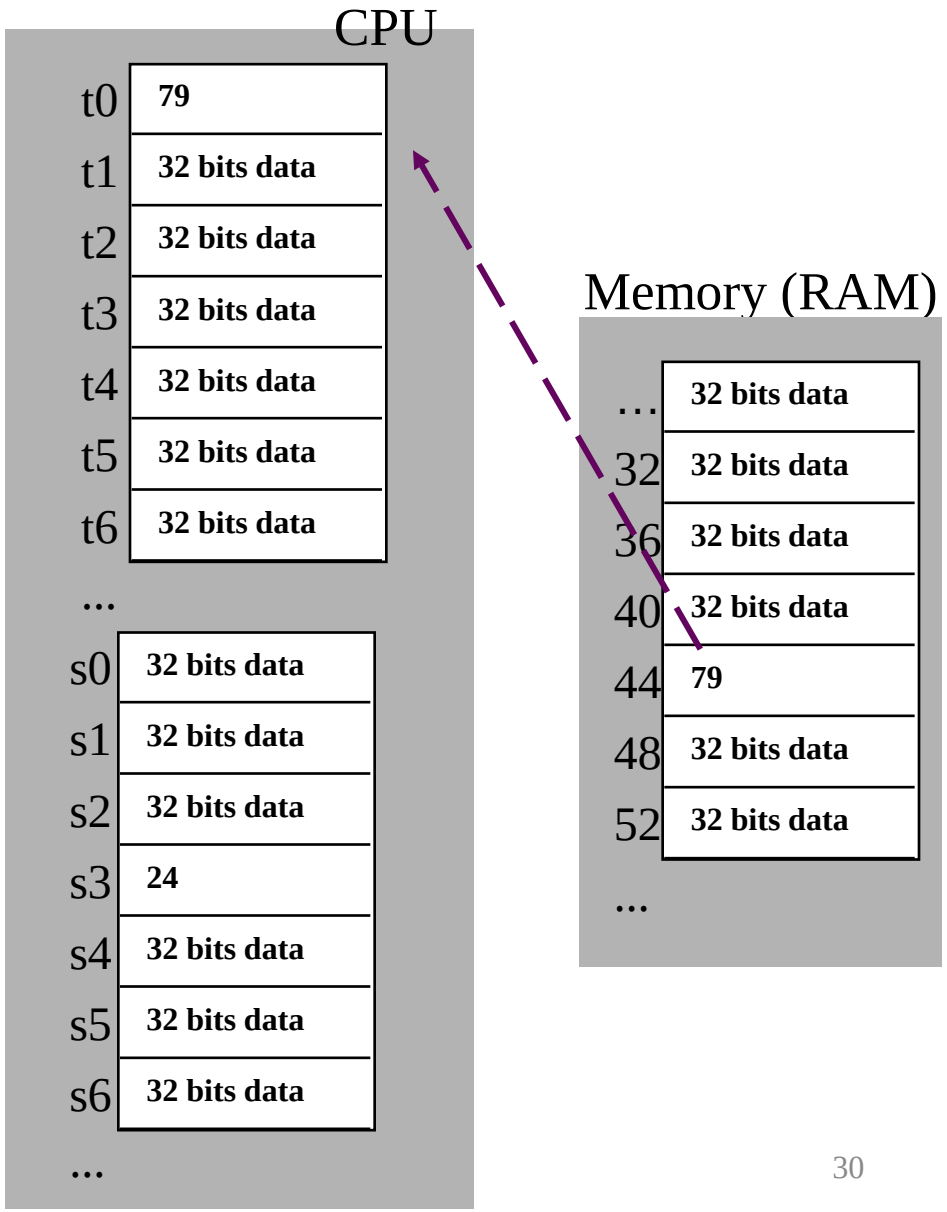
- MIPS has two basic data transfer instructions for accessing memory

```
lw $t0, 4($s3) #load word from memory
sw $t0, 8($s3) #store word to memory
```

- The data is loaded into (lw) or stored from (sw) a register in the register file – a 5 bit address.
- The memory address – a 32 bit address – is formed by adding the contents of the base address register to the offset value.
  - A 16--bit field meaning access is limited to memory locations within a region of 2^{13} or 8,192 words (2^{15} or 32,768 bytes) of the address in the base register.

MIPS load and store instructions

- Load data from memory and store data to memory.
- `lw $t0, 20($s3)`
  - lw: load word.
  - Meaning: load the data at memory address [s3+20] into processor and store into register t0.
  - E.g. if s3 stores a data 24. The memory address [s3+20] is 44. The processor loads the data at memory address 44, which is 79. This data is loaded into processor and stored into t0. As a result, register t0 will store a value 79.
- `sw $t0, 20($s3) ??`
  - sw: store word.



MIPS load and store instructions

- Example:

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

  MIPS code:

```
lw  $t0, 32($s3)
add $t0, $s2, $t0
sw  $t0, 48($s3)
```

  why 32 & 48?

  32 = 4*8

  48 = 4*12

  Note:

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

  `sw $t0, 48($s3)` means `Memory[$s3+48] = $t0`

  assume `$s3` is the address of `b[0]` → base address

- Remember:
  - Arithmetic operands are registers, not memory!
  - Loading words but addressing bytes!

Example

- Can we figure out the code?

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

```
swap:
    sll $t2, $t5, 2
    add $t2, $t4, $t2
    lw $s5, 0($t2)
    lw $s6, 4($t2)
    sw $s6, 0($t2)
    sw $s5, 4($t2)
    jr $ra
```

Assume: $t5 → k

$t4 → base address of v

$t2 → address of v[k]

More MIPS data transfer instructions

Instruction	*Comment*
SW $t1, 100($t2)	Store word to location [t2 + 100]
SH $t1, 102($t2)	Store lower halfword of t1 to location [t2+102]
SB $t1, 103($t2)	Store least significant byte of t1 to [t2+103]
LW $t1, 100($t2)	Load word from location [t2+100]
LH $t1, 102($t2)	Load halfword with sign extend from [t2+102]
LHU $t1, 102($t2)	Load halfword unsigned from [t2+102]
LB $t1, 103($t2)	Load byte with sign extend from [t2+103]
LBU $t1, 103($t2)	Load byte unsigned from [t2+103]

MIPS control based instructions

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

- MIPS conditional branch instructions:

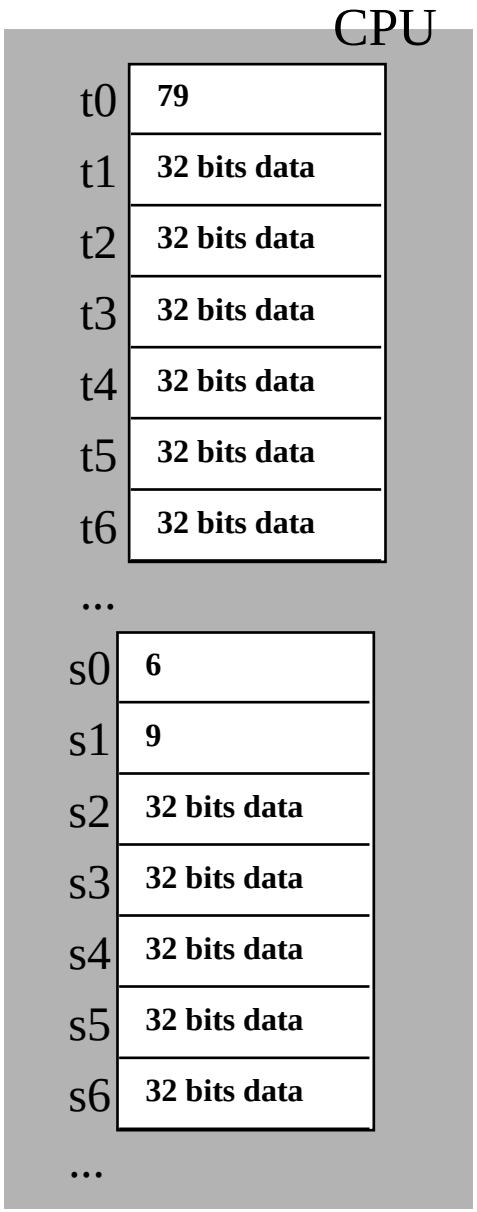
```
bne $t0, $t1, Label
beq $t0, $t1, Label
```

- bne: branch on not equal
  - The program jumps to “Label” when the values of t0 and t1 are not equal.
- beq: branch on equal
  - The program jumps to “Label” when the values of t0 and t1 are equal.

MIPS control based instructions

- When the processor executes the following assembly program. Since the value of s0 is 6, and the value of s1 is 9, it means they are not equal. The program jumps from “bne” to “abc”, where the instruction “sub” is executed and instruction “add” is ignored. After “sub” is executed, “lw” will be executed.

```
        bne $s0, $s1, abc
        add $s3, $s0, $s1
abc:    sub $s4, $s1, $s2
        lw $t0, 8($t1)
        ...
```



MIPS control based instructions

- Example:

```
if (i==j) {
    h = i + j;
}
k = j – f;
```

```
      bne $s0, $s1, abc
      add $s3, $s0, $s1
abc:  sub $s4, $s1, $s2
      ...
```

MIPS control based instructions

- MIPS unconditional branch instructions: jump to “Label” directly.

```
j  Label
```

- Example 1:

```
abc: a = b + c;
     goto abc;
```

```
abc: add $s1, $s2, $s3
     j abc;
```

- Example 2:

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

```
    beq $s4, $s5, abc1
    add $s3, $s4, $s5
    j   abc2
abc1: sub $s3, $s4, $s5
abc2: ...
```

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

In Support of Branch Instructions

- We have beq, bne, but what about other kinds of branches (e.g., branch--if--less--than)? For this, we need yet another instruction, `slt`
- Set on less than instruction:

```
slt $t0, $s0, $s1 # if $s0 < $s1 then
                  # $t0 = 1 else
                  # $t0 = 0
```

- Instruction format (R format)
- Alternate versions of slt

```
slti $t0, $s0, 25  # if $s0 < 25 then $t0=1 ...
sltu $t0, $s0, $s1 # if $s0 < $s1 then $t0=1 ...
sltiu $t0, $s0, 25 # if $s0 < 25 then $t0=1 ...
```

Signed vs. Unsigned Comparison

		Signed	Unsigned
R1=	0…00 0000 0000 0000 0001	1_{10}	1_{10}
R2=	0…00 0000 0000 0000 0010	2_{10}	2_{10}
R3=	1…11 1111 1111 1111 1111	-1_{10}	4294967295_{10}

- After executing these instructions:

```
slt  r4,r2,r1 ; if (r2 < r1) r4=1 else r4=0
slt  r5,r3,r1 ; if (r3 < r1) r5=1 else r5=0
sltu r6,r2,r1 ; if (r2 < r1) r6=1 else r6=0
sltu r7,r3,r1 ; if (r3 < r1) r7=1 else r7=0
```

- What are values of registers r4 - r7? Why?

```
r4 =  0  ; r5 =  1  ; r6 =  0  ; r7 =  0  ;
```

Aside: More Branch Instructions

- Can use `slt`, `beq`, `bne`, and the fixed value of 0 in register `$zero` to create other conditions
  - less than `blt $s1, $s2, Label`

```
slt $at, $s1, $s2 #$at set to 1 if
bne $at, $zero, Label #$s1 < $s2
```

  - less than or equal to `ble $s1, $s2, Label`
  - greater than `bgt $s1, $s2, Label`
  - great than or equal to `bge $s1, $s2, Label`
- Such branches are included in the instruction set as pseudo instructions -- recognized (and expanded) by the assembler.
  - Its why the assembler needs a reserved register ($at).

Bounds Check Shortcut

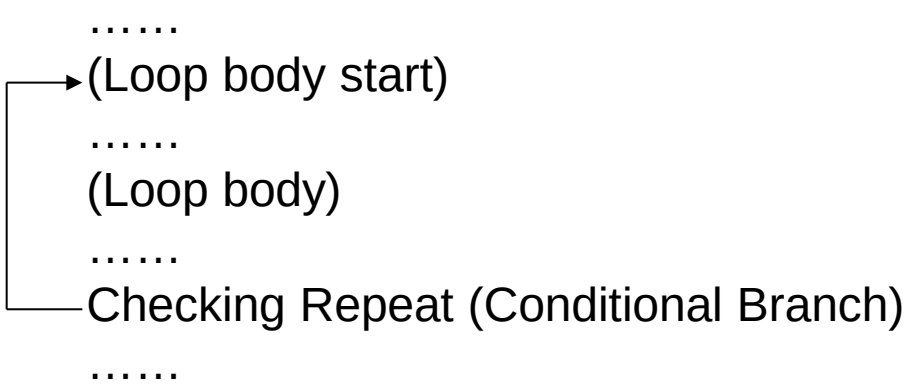
- Treating signed numbers as if they were unsigned gives a low cost way of checking if 0 ≤ x < y (index out of bounds for arrays)

```
sltu $t0, $s1, $t2      # $t0 = 0 if
                        # $s1 > $t2 (max)
                        # or $s1 < 0 (min)
beq $t0,$zero,IOOB      # go to IOOB if
                        # $t0 = 0
```

- The key is that negative integers in two's complement look like large numbers in unsigned notation. Thus, an unsigned comparison of x < y also checks if x is negative as well as if x is less than y.

Looping using compare & branch instructions

- Do-while loop

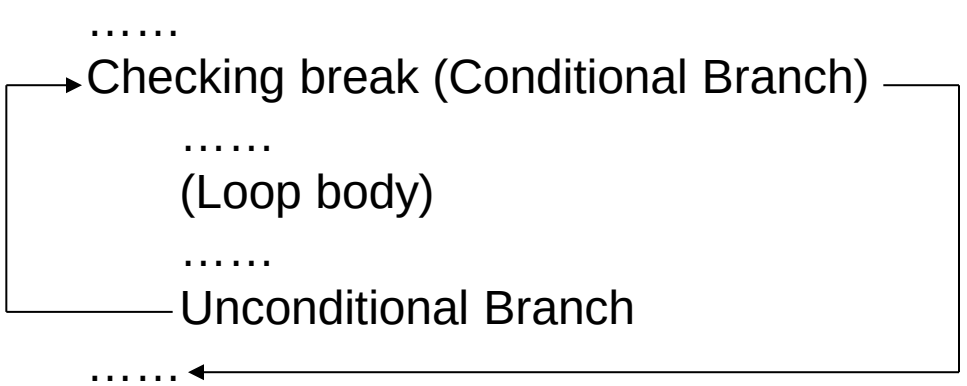



```
i = 0;
do {
   do something;
   i = i + 1;
} while (i < 10)
```

```
          addi $s1, $zero, 0
START:    do something;
          addi $s1, $s1, 1
          slti $t1, $s1, 10
          bne $t1, $zero, START
```

Looping using compare & branch instructions

- While loop

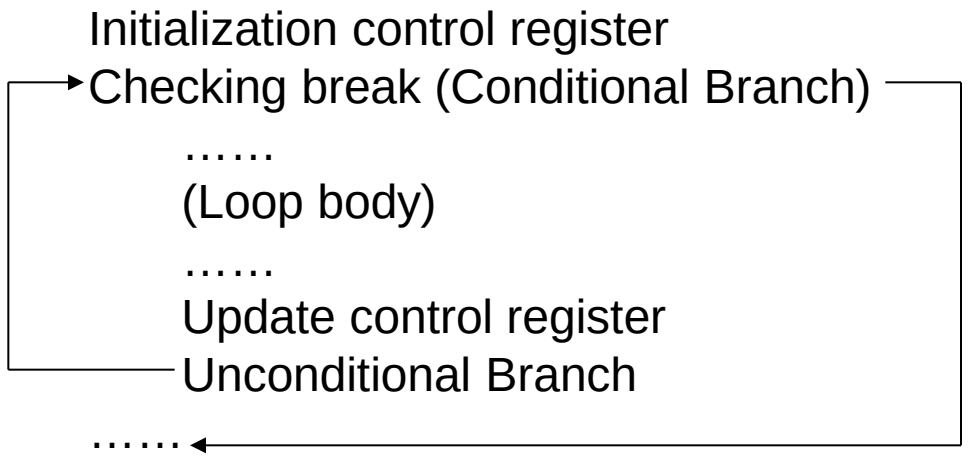



```
i = 0;
while (i < 10) {
    do something;
    i = i + 1;
}
```

```
            addi $s1, $zero, 0
            j CHECK
START:      do something;
            addi $s1, $s1, 1
CHECK:      slti $t1, $s1, 10
            bne $t1, $zero, START
```

Looping using compare & branch instructions

- FOR loop




```
for (i = 0; i < 10; i++)
{
    do something;
}
```

```
        addi $s1, $zero, 0
START:  slti $t1, $s1, 10
        beq $t1, $zero, QUIT
        do something;
        addi $s1, $s1, 1
        j START
QUIT:   ....
```

Alternative implementation of FOR

```
for (i = 0; i < 10; i++)
{
    do something;
}
```

```
        addi $s1, $zero, 0
        j CHECK
START:  do something;
        addi $s1, $s1, 1
CHECK:  slti $t1, $s1, 10
        bne $t1, $zero, START
```

MIPS arithmetic instructions

Instruction	*Example*	*Meaning*	*Comments*
add	add $t1,$t2,$t3	$t1 = $t2 + $t3	3 operands;
subtract	sub $t1,$t2,$t3	$t1 = $t2 – $t3	3 operands;
add immediate	addi $t1,$t2,100	$t1 = $t2 + 100	+ constant;
add unsigned	addu $t1,$t2,$3	$t1 = $t2 + $t3	3 operands;
subtract unsigned	subu $t1,$t2,$3	$t1 = $t2 – $t3	3 operands;
add imm. unsign.	addiu $t1,$t2,100	$t1 = $t2 + 100	+ constant;
multiply	mult $t2,$t3	Hi, Lo = $t2 x $t3	64-bit signed product
multiply unsigned	multu $t2,$t3	Hi, Lo = $t2 x $t3	64-bit unsigned product
divide	div $t2,$t3	Lo = $t2 ÷ $t3, Hi = $t2 mod $t3	Lo = quotient, Hi = remainder
divide unsigned	divu $t2,$t3	Lo = $t2 ÷ $t3, Hi = $t2 mod $t3	Unsigned quotient & remainder
Move from Hi	mfhi $t1	$t1 = Hi	Used to get copy of Hi
Move from Lo	mflo $t1	$t1 = Lo	Used to get copy of Lo

MIPS bitwise and logical instructions

Instruction	*Example*	*Meaning*	*Comment*
and	and $t1,$t2,$t3	$t1 = $t2 & $t3	3 reg. operands; Logical AND
or	or $t1,$t2,$t3	$t1 = $t2 | $t3	3 reg. operands; Logical OR
xor	xor $t1,$t2,$t3	$t1 = $t2 ⊕ $t3	3 reg. operands; Logical XOR
nor	nor $t1,$t2,$t3	$t1 = ~($t2 |$t3)	3 reg. operands; Logical NOR
and immediate	andi $t1,$t2,10	$t1 = $t2 & 10	Logical AND reg, constant
or immediate	ori $t1,$t2,10	$t1 = $t2 | 10	Logical OR reg, constant
xor immediate	xori $t1, $t2,10	$t1 = $t2 ⊕10	Logical XOR reg, constant
shift left logical	sll $t1,$t2,10	$t1 = $t2 << 10	Shift left by constant
shift right logical	srl $t1,$t2,10	$t1 = $t2 >> 10	Shift right by constant
shift right arithm.	sra $t1,$t2,10	$t1 = $t2 >> 10	Shift right (sign extend)
shift left logical	sllv $t1,$t2,$t3	$t1 = $t2 << $t3	Shift left by variable
shift right logical	srlv $t1,$t2, $t3	$t1 = $t2 >> $t3	Shift right by variable
shift right arithm.	srav $t1,$t2, $t3	$t1 = $t2 >> $t3	Shift right arith. by variable

MIPS jump, branch, compare instructions

Instruction	*Example*	*Meaning*
branch on equal	beq $t1,$t2,abc	if ($t1 == $t2) jump to PC+4+imm×4
	Equal test; PC relative branch	
branch on not eq.	bne $t1,$t2,abc	if ($t1!= $t2) jump to PC+4+imm×4
	Not equal test; PC relative	
set on less than	slt $t1,$t2,$t3	if ($t2 < $t3) $t1=1; else $t1=0
	Compare less than; $t2 and $t3 are 2's complement	
set less than imm.	slti $t1,$t2,100	if ($t2 < 100) $t1=1; else $t1=0
	Compare < constant; $t2 is 2's complement	
set less than uns.	sltu $t1,$t2,$t3	if ($t2 < $t3) $t1=1; else $t1=0
	Compare less than; $t2 and $t3 are natural numbers	
set l. t. imm. uns.	sltiu $t1,$t2,100	if ($t2 < 100) $t1=1; else $t1=0
	Compare < constant; $t2 is natural number	
jump	j Label	jump to Label
jump and link	jal Label	$t31 = PC + 4; jump to Label
	For procedure call, $t31 stores the return address	
jump register	jr $ra	jump to a location specified by $ra
	For procedure return	

MIPS: Software conventions for Registers

0	zero	constant 0
1	at	reserved for assembler
2	v0	expression evaluation &
3	v1	function results
4	a0	arguments (proc. call)
5	a1	
6	a2	
7	a3	
8	t0	temporary
. . .		(callee can clobber)
15	t7	
16	s0	preserved, callee
		saves and restore
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 (HW)

Why do we need register backup?

```
int a = 10;
void main() {
    int b=1;
    b = addOne(b);
    a = a + b;
}
int addOne(int c) {
    int b;
    a = 1;
    b = a + c;
    return b;
}
```

```
int a = 10;
void main() {
    int b=1;
    b = addOne(b);
    a = a + b;
}
int addOne(int c) {
    int tmp;
    int b;
    tmp = a;
    a = 1;
    b = a + c;
    a = tmp;
    return b;
}
```

Instructions for Accessing Procedures

- MIPS procedure call instruction:

```
jal ProcedureAddress #jump and link
```

- Saves PC+4 in register $ra to have a link to the next instruction for the procedure return.
  - Instruction format (J format)

- Then can do procedure return with a

```
jr $ra #return
```

  - Instruction format (R format)

Six Steps in Execution of a Procedure

- 1. Main routine (caller) places parameters in a place where the procedure (callee) can access them
  - $a0 -- $a3: four argument registers
- 2. Caller transfers control to the callee
- 3. Callee acquires the storage resources needed
- 4. Callee performs the desired task
- 5. Callee places the result value in a place where the caller can access it
  - $v0 -- $v1: two value registers for result values
- 6. Callee returns control to the caller
  - $ra: one return address register to return to the point of origin

Register backup in Procedure Call

- Some compilers can generate it automatically.
- If you write your own assembly program, you need to handle it manually.

Procedure Call

- Caller
  - Backup registers
  - Place arguments where procedure can access them ($a0..$a3, or stack)
  - Call procedure (execute a “jal” instruction)
  - Collect result from stack or registers
  - Restore registers
- Callee
  - Backup registers
  - Get the arguments from stack or registers ($a0-$a3)
  - Do the work
  - Save result to stack or registers
  - Restore registers
  - Return

Basic Procedure Flow

- For a procedure that computes the GCD of two values `i` (in `$t0`) and j (in `$t1`)

```
gcd(i,j);
```

- The caller puts the `i` and `j` (the parameters values) in `$a0` and `$a1` and issues a

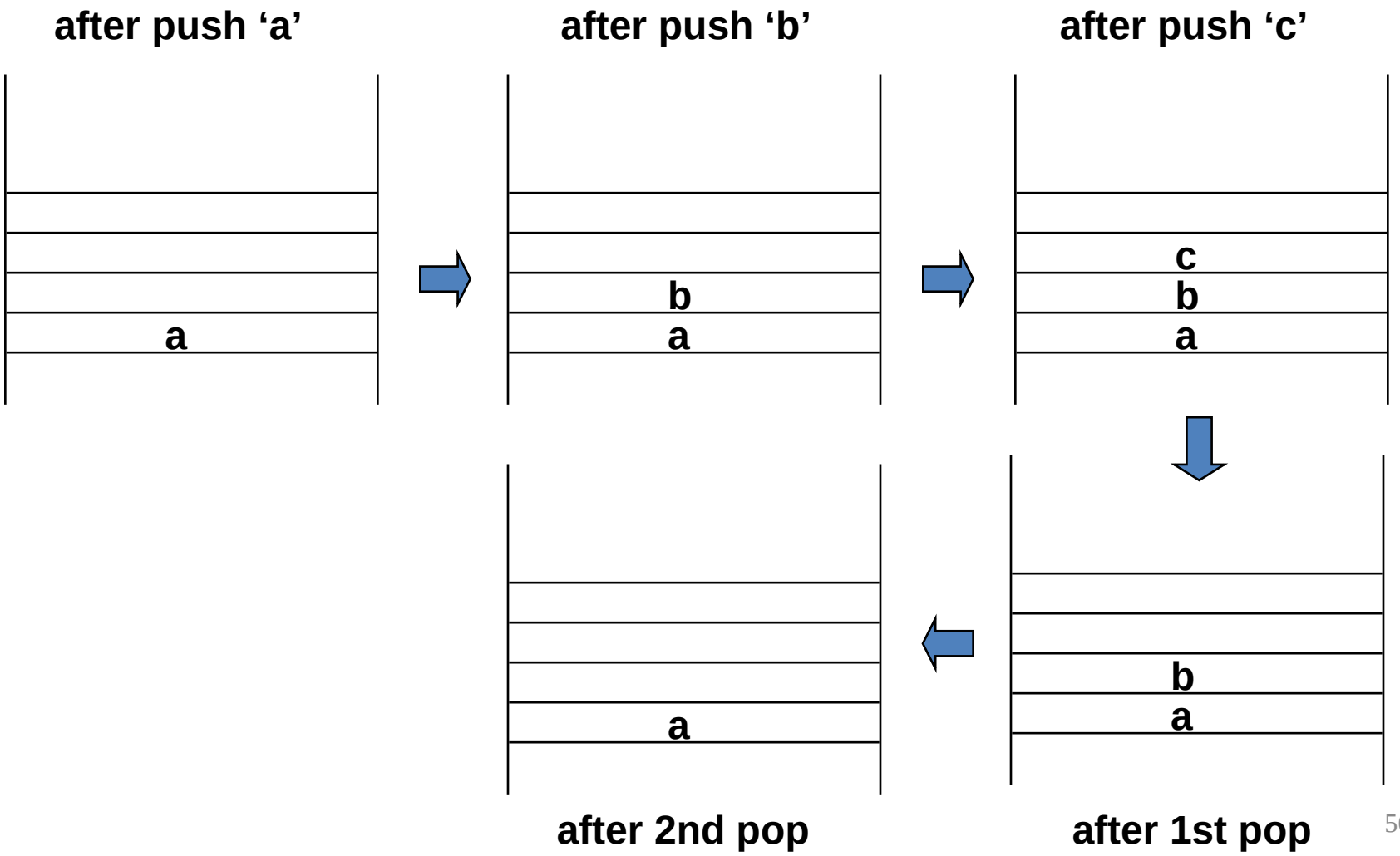
```
jal gcd  #jump to routine gcd
```

- The callee computes the GCD, puts the result in `$v0`, and returns control to the caller using

```
gcd: . . .   #code to compute gcd
     jr $ra  #return
```

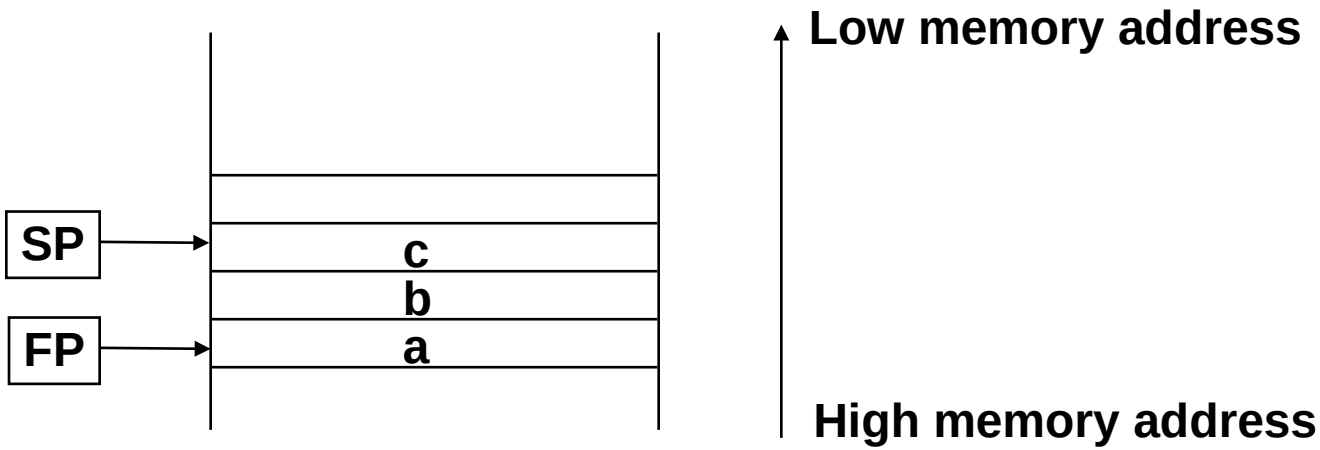
Stack

- Last in first out, first in last out



Memory Stacks

- MIPS: software convention, i.e. stack is implemented by memory.
- SP (Stack pointer): a special register points to the top of stack, i.e. the address of the first element in the stack.
- FP (Frame pointer): a special register points to the bottom of stack.

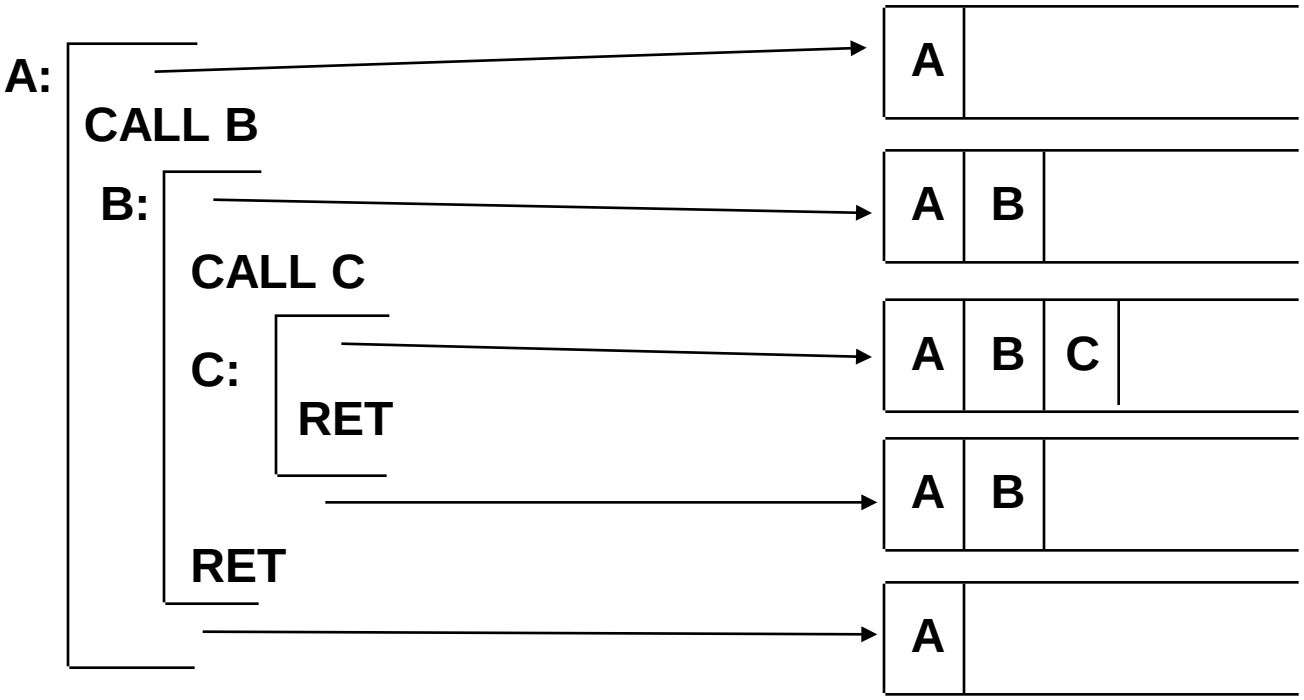


- **Push data to stack**
  - **decrement SP**
  - **store data to the place pointed by SP**

- **Pop data from stack**
  - **load data pointed by SP**
  - **increment SP**

Calls: Why Are Stacks So Great?

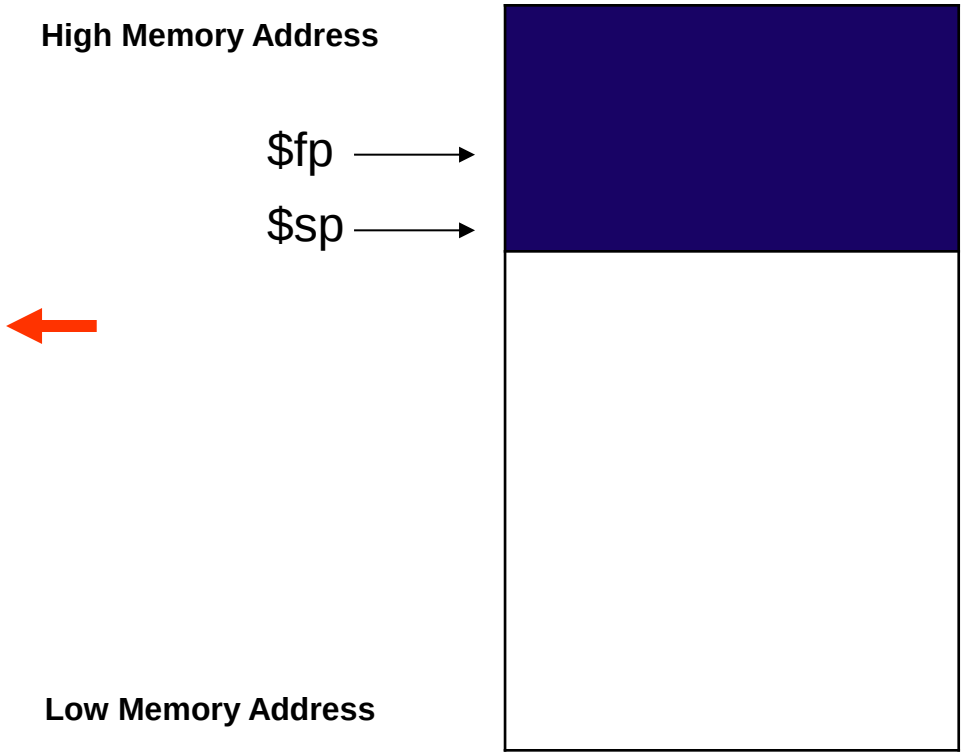
Stacking of Subroutine Calls & Returns and Environments:



Example

- Before Calling Procedure

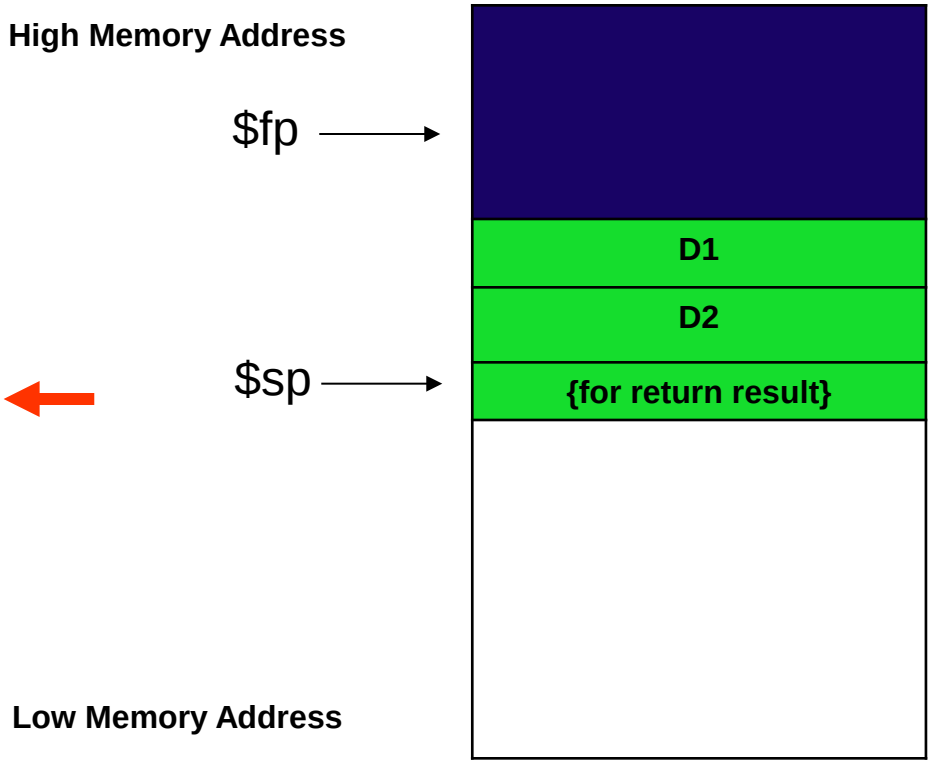
```
Main:
    ……
    RESULT = abc(D1, D2);
    ……
```



Example

- Passing arguments

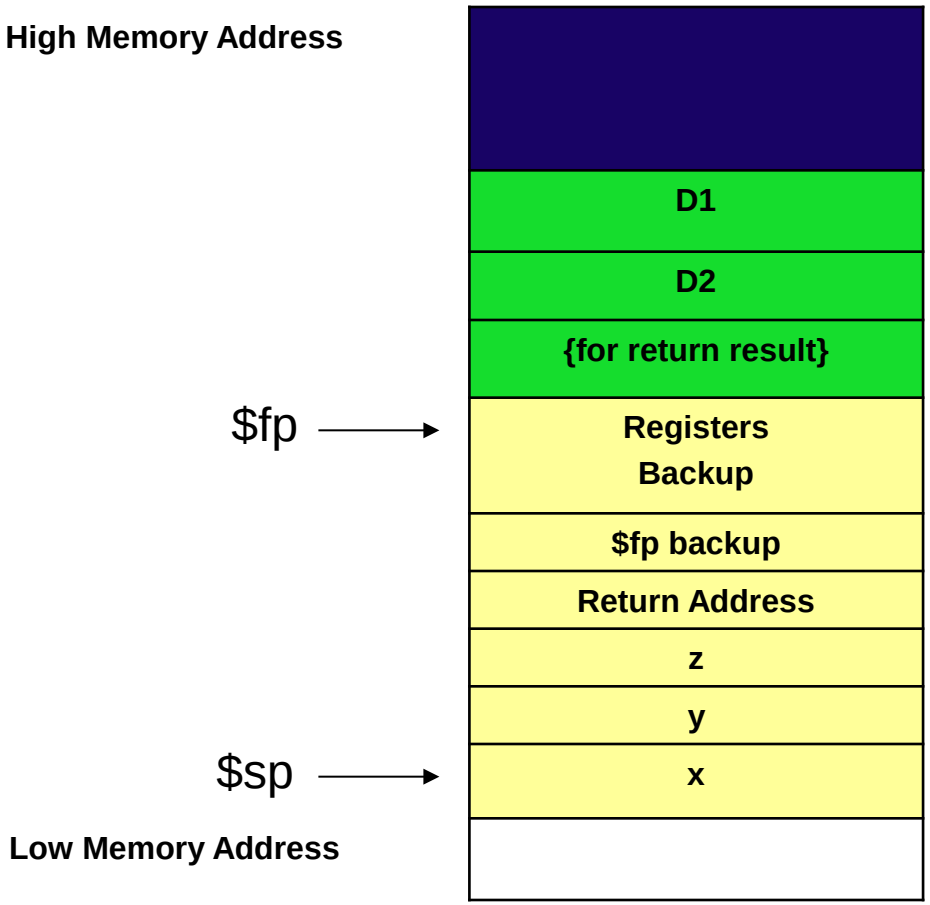
```
Main:
    ……
    RESULT = abc(D1, D2);
    ……
```



Example

- Before executing procedure body

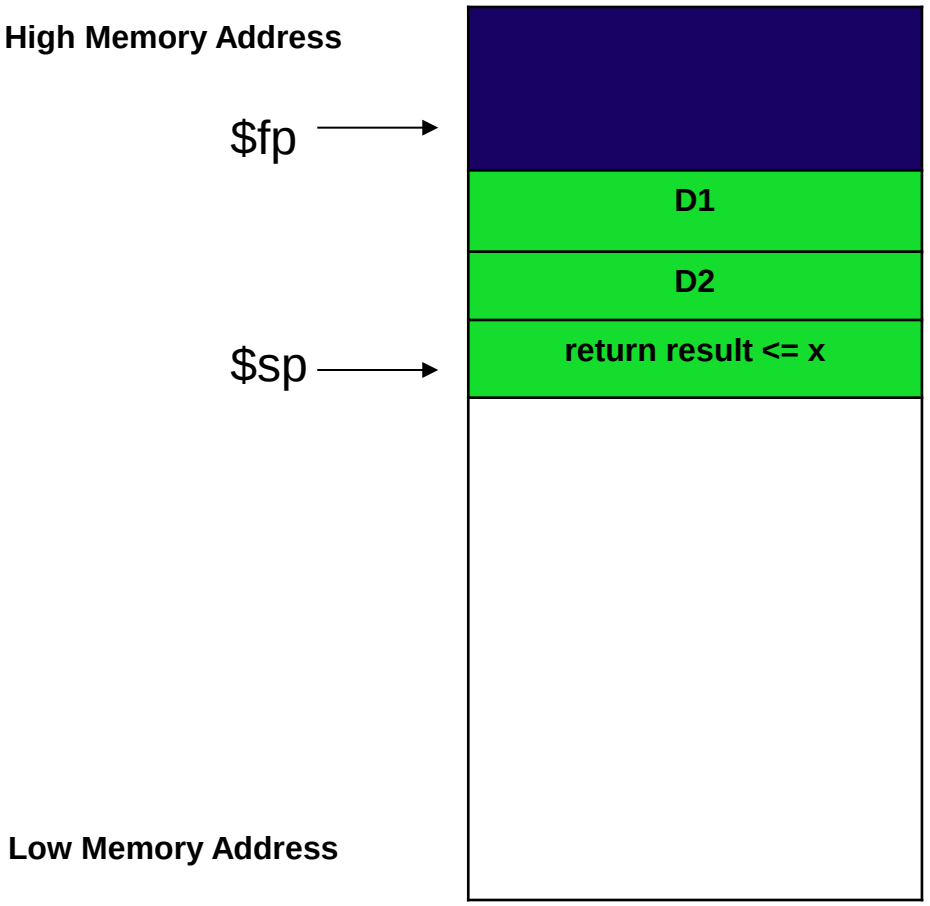
```
int abc(int par1, int par2)
{
        int x, y, z;
        ……
        ……
        return x;
}
```



Example

- Before returning to caller

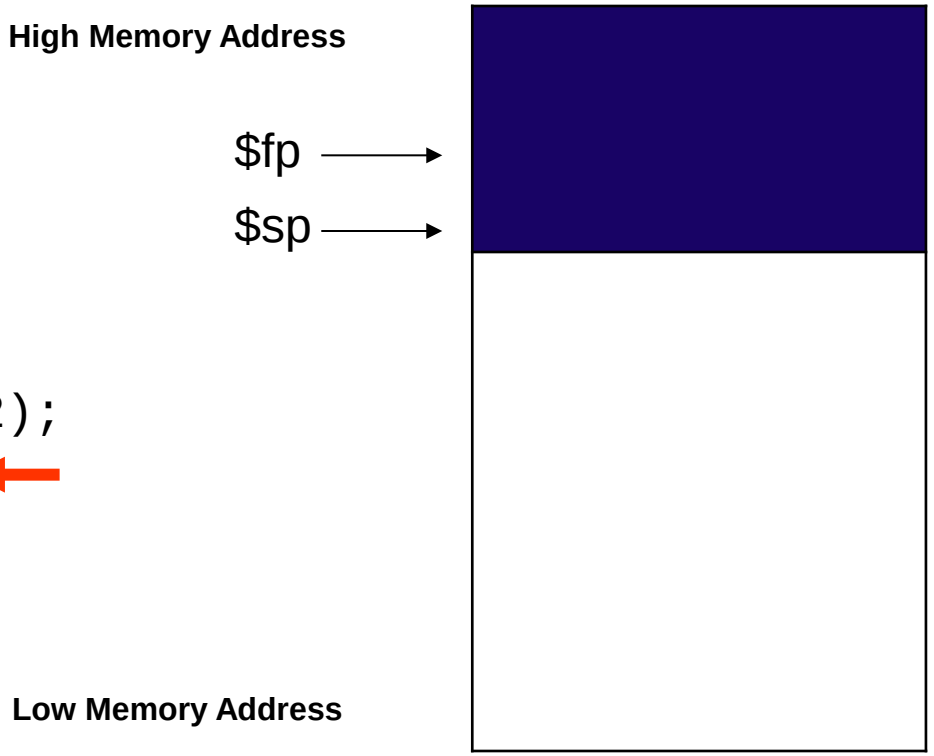
```
int abc(int par1, int par2)
{
        int x, y, z;
        ……
        ……
        return x;
}
```



Example

- After Calling Procedure

```
Main:
    ……
    RESULT = function(D1, D2);
    ……
```



Compiling a C Leaf Procedure

- Leaf procedures are ones that do not call other procedures. Give the MIPS assembler code for

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

where `g`, `h`, `i`, and `j` are in `$a0`, `$a1`, `$a2`, `$a3`

```
leaf_ex:   addi $sp,$sp,-8      #make stack room
           sw $t1,4($sp)        #save $t1 on stack
           sw $t0,0($sp)        #save $t0 on stack
           add $t0,$a0,$a1
           add $t1,$a2,$a3
           sub $v0,$t0,$t1
           lw $t0,0($sp)        #restore $t0
           lw $t1,4($sp)        #restore $t1
           addi $sp,$sp,8       #adjust stack ptr
           jr $ra
```

Nested Procedures

- What happens to return addresses with nested procedures?

```
int rt_1 (int i) {
    if (i == 0) return 0;
        else return rt_2(i-1); }
```

```
caller: jal  rt_1
next:   . . .

rt_1:   bne  $a0, $zero, to_2
        add  $v0, $zero, $zero
        jr   $ra
to_2:   addi $a0, $a0, -1
        jal  rt_2
        jr   $ra

rt_2:   . . .
```

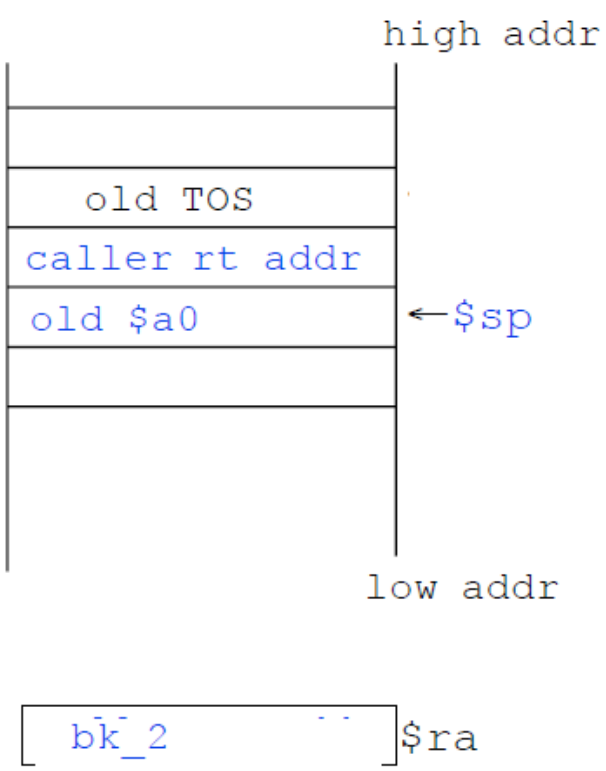
Nested Procedures Outcome

```
caller:  jal   rt_1
next:    . . .

rt_1:    bne   $a0, $zero, to_2
         add   $v0, $zero, $zero
         jr    $ra
to_2:    addi  $a0, $a0, -1
         jal   rt_2
         jr    $ra

rt_2:    . . .
```

- On the call to `rt_1`, the return address (next in the caller routine) gets stored in `$ra`. What happens to the value in `$ra` (when `$a0!=0`) when `rt_1` makes a call to `rt_2`?

Saving the Return Address, Part 1

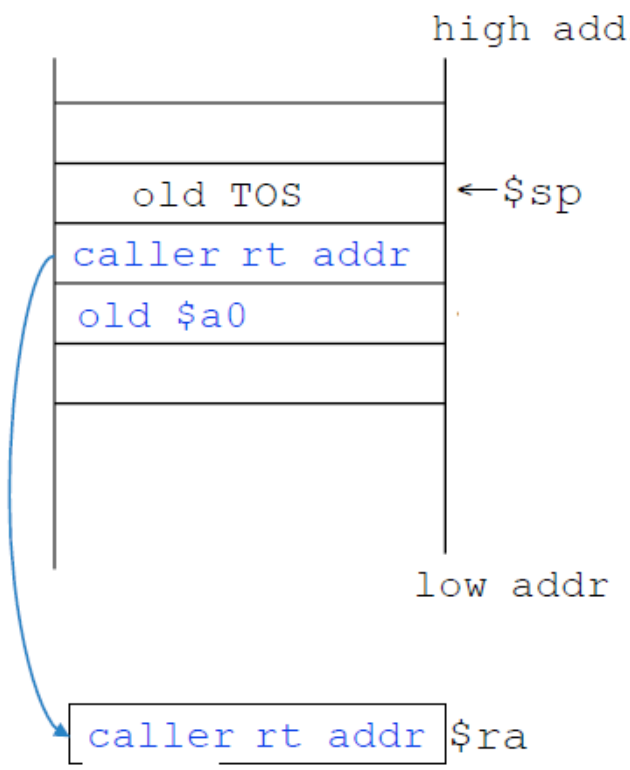
- Nested procedures (`i` passed in `$a0`, return value in `$v0`)




```
rt_1:bne  $a0, $zero, to_2
     add  $v0, $zero, $zero
     jr   $ra
to_2:addi $sp, $sp, -8
     sw   $ra, 4($sp)
     sw   $a0, 0($sp)
     addi $a0, $a0, -1
     jal  rt_2
bk_2:lw   $a0, 0($sp)
     lw   $ra, 4($sp)
     addi $sp, $sp, 8
     jr   $ra
```

Saving the Return Address, Part 2

- Nested procedures (`i` passed in `$a0`, return value in `$v0`)




```
rt_1:bne   $a0, $zero, to_2
     add   $v0, $zero, $zero
     jr    $ra
to_2:addi  $sp, $sp, -8
     sw    $ra, 4($sp)
     sw    $a0, 0($sp)
     addi  $a0, $a0, -1
     jal   rt_2
bk_2:lw    $a0, 0($sp)
     lw    $ra, 4($sp)
     addi  $sp, $sp, 8
     jr    $ra
```

Compiling a Recursive Procedure

- A procedure for calculating factorial

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

- A recursive procedure (one that calls itself!)

fact (0) = 1

fact (1) = 1 * 1 = 1

fact (2) = 2 * 1 * 1 = 2

fact (3) = 3 * 2 * 1 * 1 = 6

fact (4) = 4 * 3 * 2 * 1 * 1 = 24

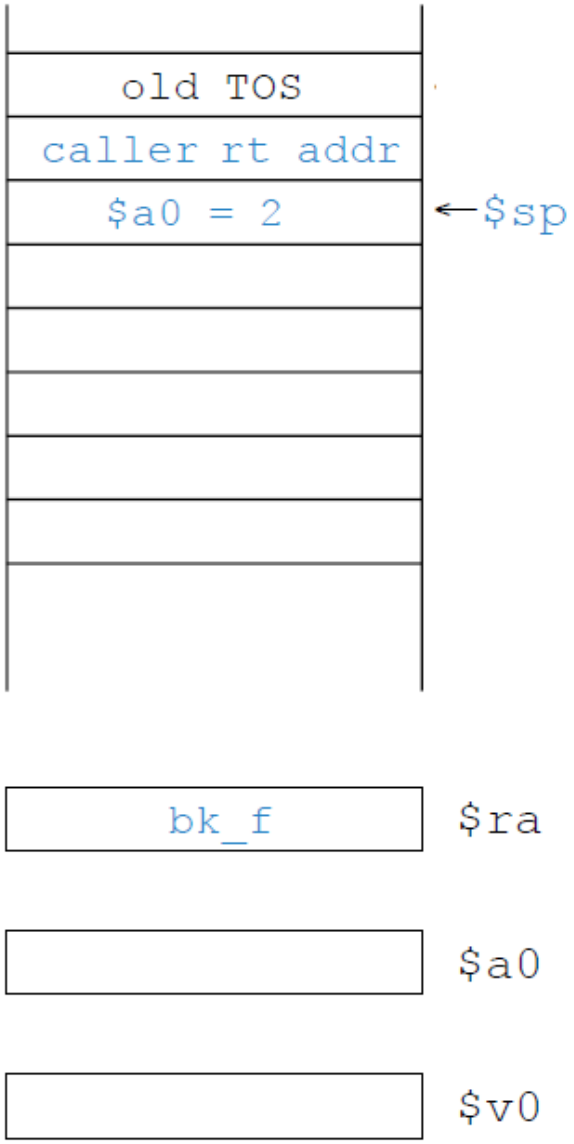
. . .

- Assume n is passed in `$a0`; result returned in `$v0`

Compiling a Recursive Procedure

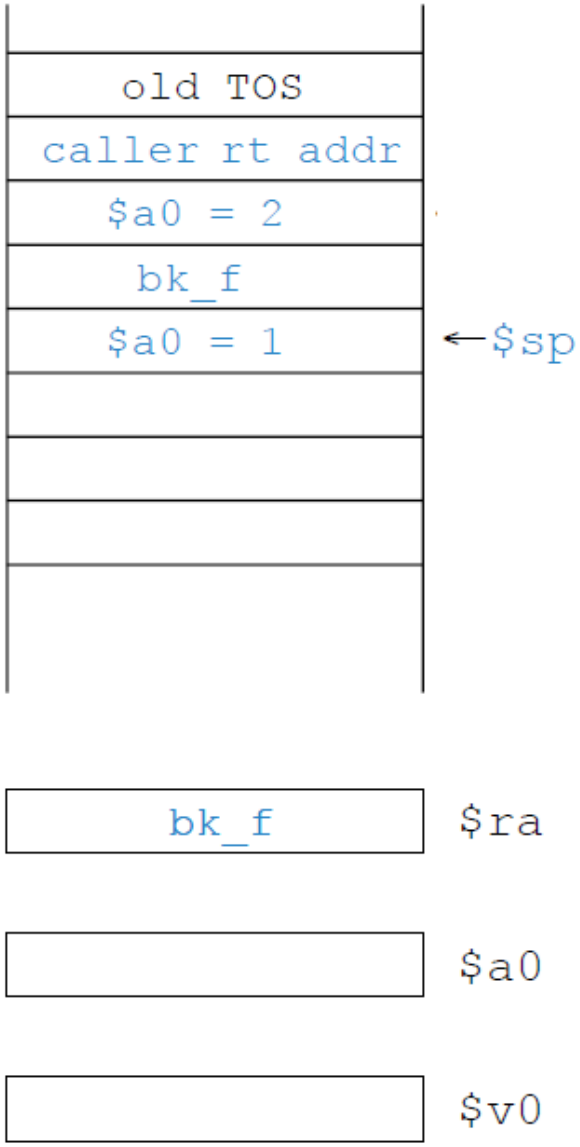
```
fact: addi $sp, $sp, -8        #adjust stack pointer
      sw $ra, 4($sp)           #save return address
      sw $a0, 0($sp)           #save argument n
      slti $t0, $a0, 1         #test for n < 1
      beq $t0, $zero, L1       #if n >=1, go to L1
      addi $v0, $zero, 1       #else return 1 in $v0
      addi $sp, $sp, 8         #adjust stack pointer
      jr $ra                   #return to caller
L1:   addi $a0, $a0, -1        #n >=1, so decrement n
      jal fact                 #call fact with (n-1)
                               #this is where fact
                               #returns
bk_f: lw $a0, 0($sp)           #restore argument n
      lw $ra, 4($sp)           #restore return address
      addi $sp, $sp, 8         #adjust stack pointer
      mul $v0, $a0, $v0        #$v0 = n * fact(n-1)
      jr $ra                   #return to caller
```

A Look at the Stack for $a0 = 2, Part 1



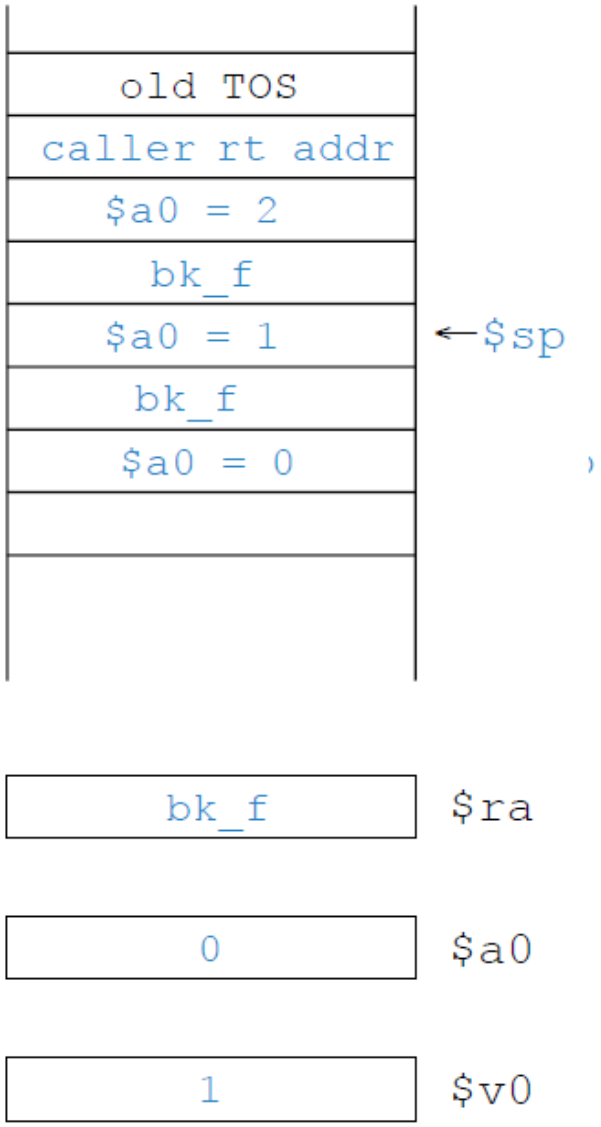
- Stack state after execution of first encounter with the `jal` instruction (*second* call to fact routine with `$a0` now holding 1)
  - saved return address to caller routine (i.e., location in the main routine where *first* call to fact is made) on the stack
  - saved original value of `$a0` on the stack

A Look at the Stack for $a0 = 2, Part 2



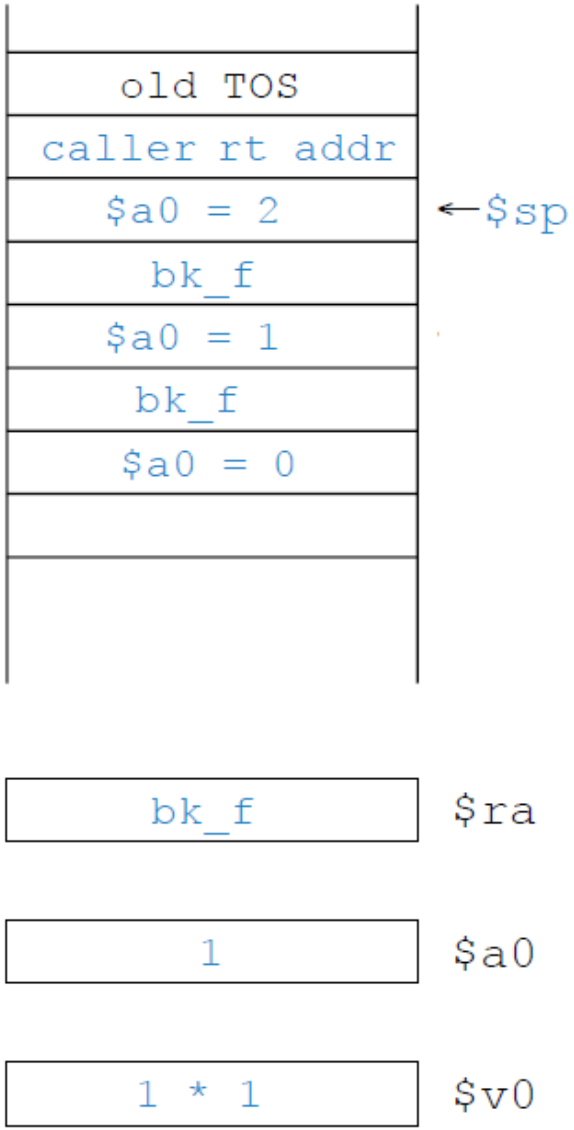
- Stack state after execution of second encounter with the `jal` instruction (*third* call to fact routine with `$a0` now holding 0)
  - saved return address of instruction in caller routine (instruction after `jal`) on the stack
  - saved previous value of `$a0` on the stack

A Look at the Stack for $a0 = 2, Part 3



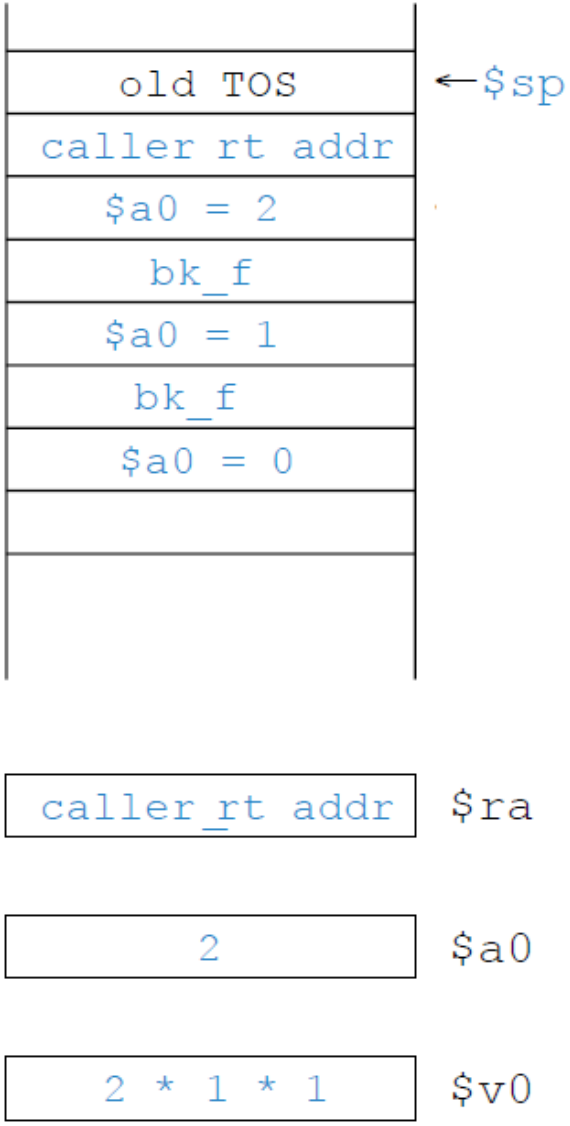
- Stack state after execution of first encounter with the first `jr` instruction (`$v0` initialized to 1)
  - stack pointer updated to point to *third* call to fact

A Look at the Stack for $a0 = 2, Part 4



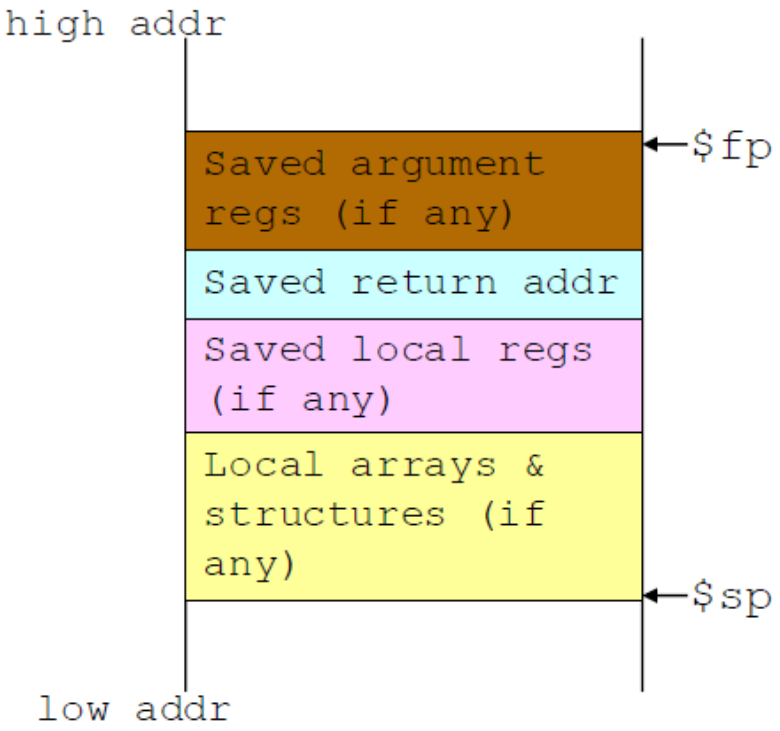
- Stack state after execution of first encounter with the second `jr` instruction (return from fact routine after updating `$v0` to 1 * 1)
  - return address to caller routine (`bk_f` in fact routine) restored to `$ra` from the stack
  - previous value of `$a0` restored from the stack
  - stack pointer updated to point to *second* call to fact

A Look at the Stack for $a0 = 2, Part 5



- Stack state after execution of second encounter with the second `jr` instruction (return from fact routine after updating `$v0` to 2 * 1 * 1)
  - return address to caller routine (main routine) restored to `$ra` from the stack
  - original value of `$a0` restored from the stack
  - stack pointer updated to point to *first* call to fact

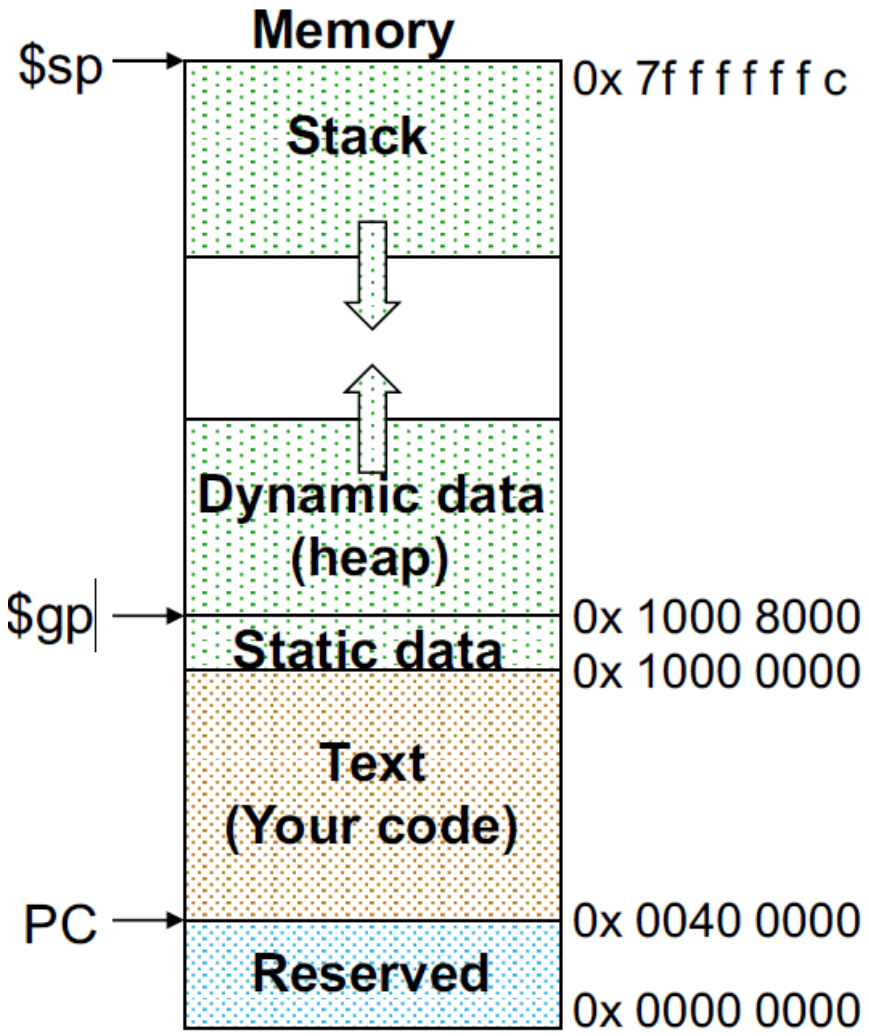
Aside: Allocating Space on the Stack



- The segment of the stack containing a procedure's saved registers and local variables is its procedure frame (aka activation record)
  - The frame pointer ($fp) points to the first word of the frame of a procedure – providing a stable "base" register for the procedure
    - $fp is initialized using $sp on a call and $sp is restored using $fp on a return

Aside: Allocating Space on the Stack

- Static data segment for constants and other static variables (e.g., arrays)
- Dynamic data segment (aka heap) for structures that grow and shrink (e.g., linked lists)
  - Allocate space on the heap with `malloc()` and free it with `free()` in C



Assembly Language vs. Machine Language

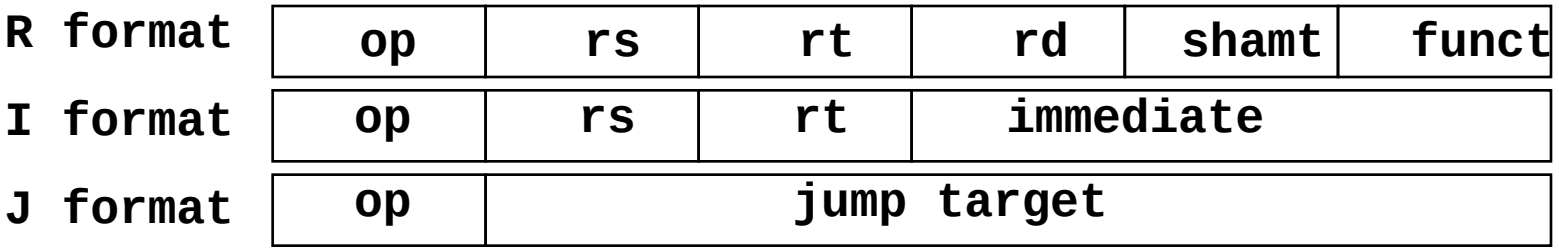
- Assembly provides convenient symbolic representation
  - easier to understand by human, e.g. add $t0, $s1, $s2
  - can provide 'pseudoinstructions', e.g., "mov $t0, $t1" exists only in Assembly would be implemented using "add $t0,$t1,$zero"
  - much easier than writing down numbers
- But machines only understand binary numbers (Machine language)
  - 10010001000 …. (It is too difficult to understand!)
- Convert assembly instructions to machine format
  - Each register has an ID: use a number to represent each register

Arithmetic, memory, branch instructions

- Instruction — Meaning

```
add $s1,$s2,$s3   $s1 = $s2 + $s3
sub $s1,$s2,$s3   $s1 = $s2 – $s3
lw $s1,100($s2)   $s1 = Memory[$s2+100]
sw $s1,100($s2)   Memory[$s2+100] = $s1
bne $s4,$s5,L     Jump to Label if $s4 != $s5
beq $s4,$s5,L     Jump to Label if $s4 = $s5
j Label           Jump to Label
```

- Divide MIPS instructions into 3 categories
- Machine formats:
  - 3 formats, all 32 bits wide
  - Very structured, rely on compiler to achieve performance



28–26 31–29	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	R-format	Bltz/gez	jump	jump & link	branch eq	branch ne	blez	bgtz
1(001)	add immediate	addiu	set less than imm.	set less than imm. unsigned	andi	ori	xori	load upper immediate
2(010)	TLB	FlPt						
3(011)								
4(100)	load byte	load half	lwl	load word	load byte unsigned	load half unsigned	lwr	
5(101)	store byte	store half	swl	store word			swr	
6(110)	load linked word	lwc1						
7(111)	store cond. word	swc1						

op(31:26)=000000 (R-format), funct(5:0)

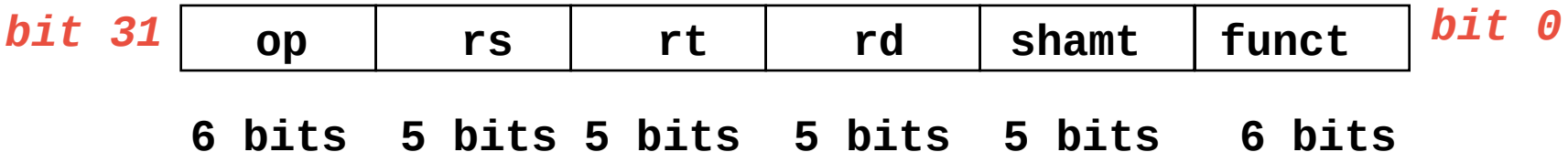
2–0 5–3	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	shift left logical		shift right logical	sra	sllv		srlv	srav
1(001)	jump register	jalr			syscall	break		
2(010)	mfhi	mthi	mflo	mtlo				
3(011)	mult	multu	div	divu				
4(100)	add	addu	subtract	subu	and	or	xor	not or (nor)
5(101)			set l.t.	set l.t. unsigned				
6(110)								
7(111)								

Aside: MIPS Register Convention

0	zero	constant 0
1	at	reserved for assembler
2	v0	expression evaluation &
3	v1	function results
4	a0	arguments (proc. call)
5	a1	
6	a2	
7	a3	
8	t0	temporary
. . .		(callee can clobber)
15	t7	
16	s0	preserved, callee
		saves and restore
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 (HW)

R type

- Mainly used to represent instructions that all their operands are registers: op rd, rs, rt
  - e.g. add $t0, $s1, $s2
  - There are exceptional cases: e.g. sll $s0, $s1, 8



- op: opcode
  - Operation of the instruction, such as add, sub, …
- rd: destination register
- rs: first source operand
- rt: second source operand
- shamt: shift amount
  - The number of bits a data being shifted/rotated
  - e.g. sll $s0, $s1, 8 → $s0 = $s1 << 8
- funct: for function that share the same opcode
  - Distinguish different operations

Example

and t0, t2, s1

op = and = 0 = 000000
rs = t2 = 10 = 01010
rt = s1 = 17 = 10001
rd = t0 = 8 = 01000
shamt = 0 = 00000
funct = 36 = 100100

op	rs	rt	rd	shamt	funct
and	t2	s1	t0	0	100100
0	10	17	8	00000	100100
000000	01010	10001	01000	00000	100100

Machine format = 0000 0001 0101 0001 0100 0000 0010 0100
0 1 5 1 4 0 2 4

Example

sll s2, s1, 8

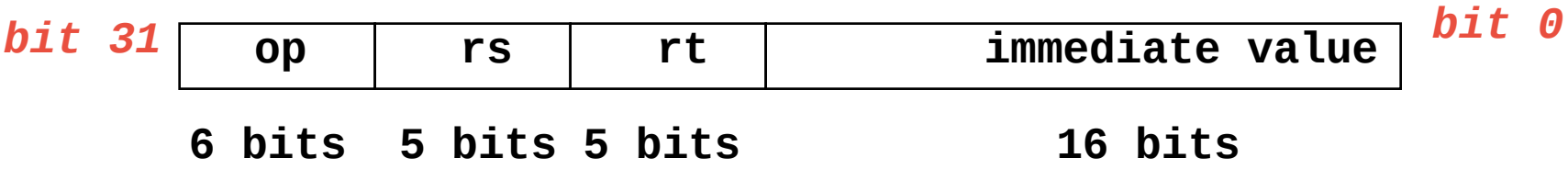
op = sll = 0 = 000000
rs = = 0 = 00000
rt = s1 = 17 = 10001
rd = s2 = 18 = 10010
shamt = 8 = 01000
funct = 0 = 000000

op	rs	rt	rd	shamt	funct
sll		s1	s2	8	000000
000000	0	17	18	01000	000000
000000	00000	10001	10010	01000	000000

Machine format = 0000 0000 0001 0001 1001 0010 0000 0000
0 0 1 1 9 2 0 0

I type

- Mainly used to represent instructions that one of their operands is a constant: op rt, rs, imm
  - e.g. addi $s0, $s1, 100



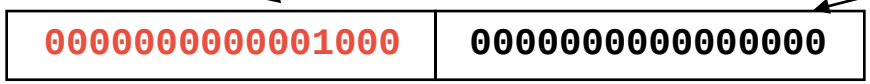
- op: opcode
  - Operation of the instruction
- rt: destination register
- rs: first source operand
- imm: immediate value, signed 16 bit number

Aside: How About Larger Constants?

- Immediate can only store 16 bit value, we'd also like to be able to load a 32 bit constant into a register,
  - E.g. 00000000000010000000000000000111
- For this we must use two instructions, a new "load upper immediate" instruction.

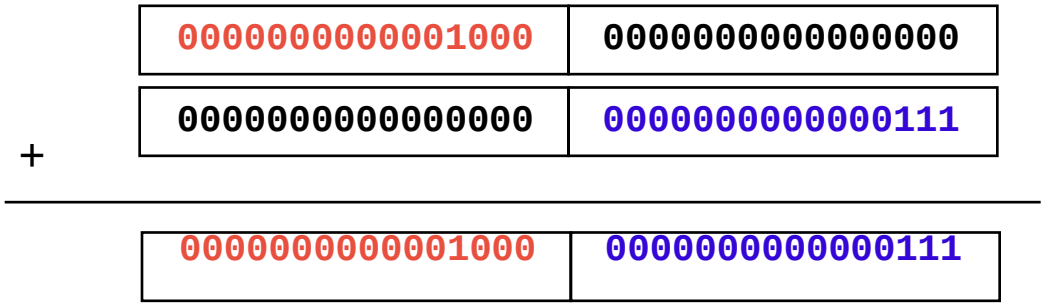
```
lui $t0, 0000000000001000
```

filled with zeros



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

```
addiu $t0, $t0, 0000000000000111
```



Example

addi t4, t3, 24

op = addi = 8 = 001000
rs = t3 = 11 = 01011
rt = t4 = 12 = 01100
imm = 24 = 0000 0000 0001 1000

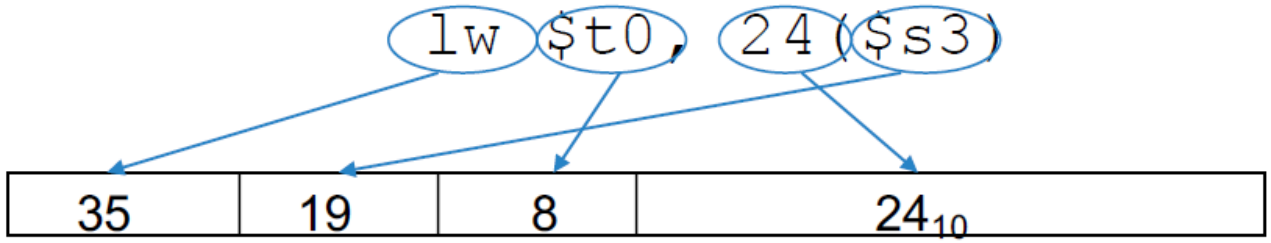
op	rs	rt	imm
andi	t3	t4	24
001000	11	12	0000 0000 0001 1000
001000	01011	01100	0000 0000 0001 1000

Machine format = 0010 0001 0110 1100 0000 0000 0001 1000

2 1 6 C 0 0 1 8

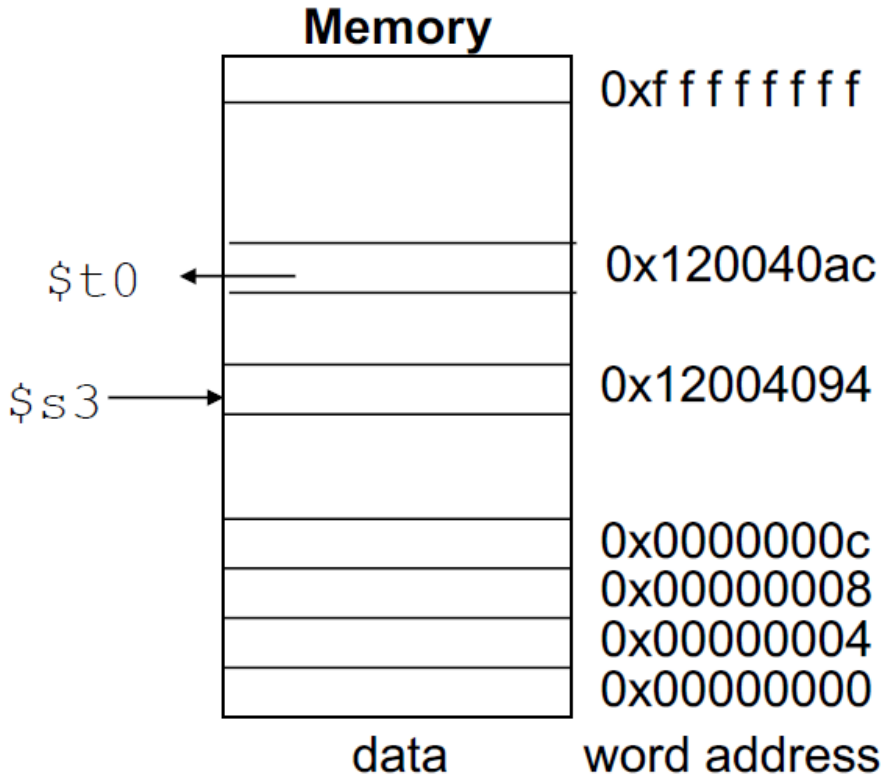
Load Instruction

- Load/Store Instruction Format (I format):



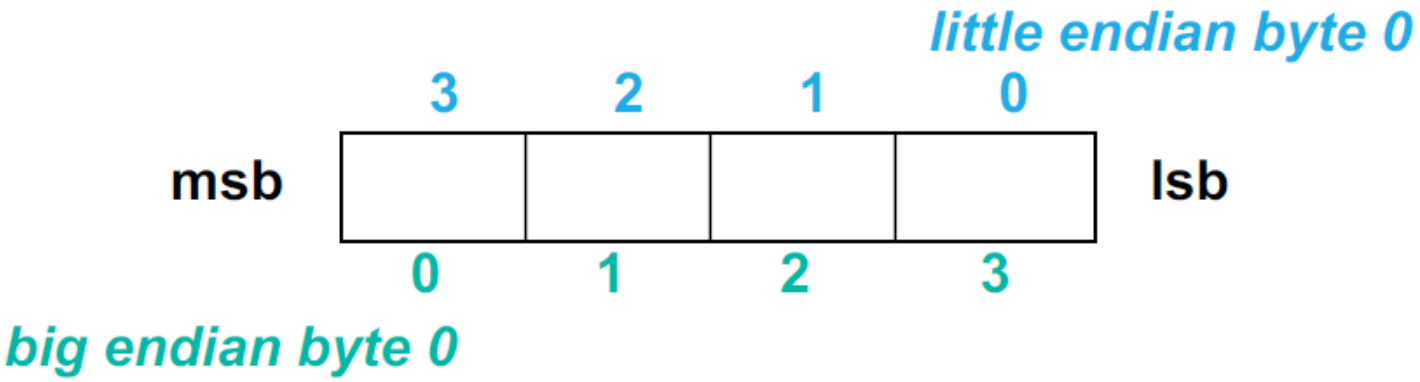
24_{10} + $s3 =

. . . 0001 1000
+ . . . 1001 0100
. . . 1010 1100 =
0x120040ac



Byte Addresses

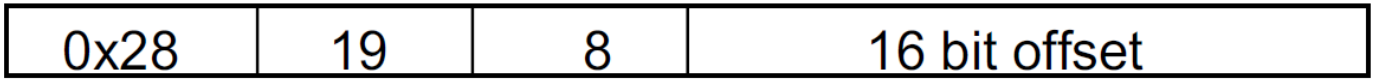
- Since 8--bit bytes are so useful, most architectures address individual bytes in memory.
  - Alignment restriction -- the memory address of a word must be on natural word boundaries (a multiple of 4 in MIPS--32).
- Big Endian: leftmost byte is word address
  - IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA
- Little Endian: rightmost byte is word address
  - Intel 80x86, DEC Vax, DEC Alpha (Windows NT)



Aside: Loading and Storing Bytes

- MIPS provides special instructions to move bytes

```
lb $t0, 1($s3) #load byte from memory
sb $t0, 6($s3) #store byte to memory
```



- What 8 bits get loaded and stored?
  - load byte places the byte from memory in the rightmost 8 bits of the destination register
    - what happens to the other bits in the register?
  - store byte takes the byte from the rightmost 8 bits of a register and writes it to a byte in memory
    - what happens to the other bits in the memory word?

EX:

- Given following code sequence and memory state what is the state of the memory after executing the code?

```
add $s3, $zero, $zero
lb $t0, 1($s3)
sb $t0, 6($s3)
```

Memory	
0x 0 0 0 0 0 0 0 0	24
0x 0 0 0 0 0 0 0 0	20
0x 0 0 0 0 0 0 0 0	16
0x 1 0 0 0 0 0 1 0	12
0x 0 1 0 0 0 4 0 2	8
0x F F F F F F F F	4
0x 0 0 9 0 1 2 A 0	0

- What value is left in $t0?
- What word is changed in Memory and to what?
- What if the machine was little Endian?

Example

lw s4, 4(s3)

op = lw		= 100011
rs = s3	= 19	= 10011
rt = s4	= 20	= 10100
imm	= 4	= 0000 0000 0000 0100

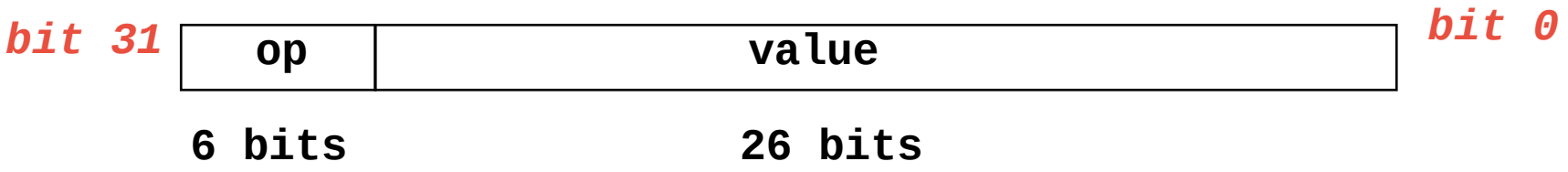
op	rs	rt	imm
lw	s3	s4	4
100011	19	20	0000 0000 0000 0100
100011	10011	10100	0000 0000 0000 0100

Machine format = 1000 1110 0111 0100 0000 0000 0000 0100

4 E 7 4 0 0 0 4

J type

- Mainly used for jump instructions: op label
  - e.g. j Label



- op: opcode
  - Operation of the instruction, e.g. j
- label: 26-bit address

Addresses in Branches and Jumps

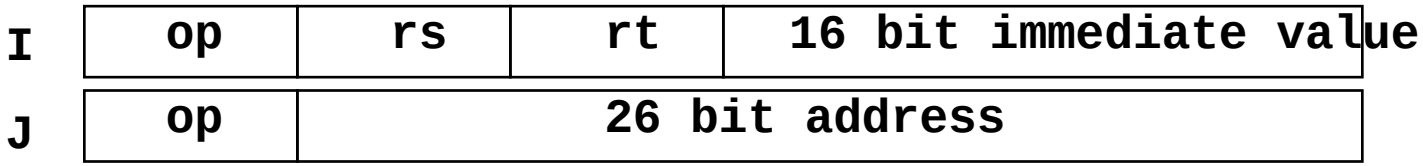
- Instructions:

  `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:



- What value shall we use to represent Label?

Conditional branch

assembly code:

```
        addi $t0,$zero,9
        addi $t1,$zero,10
        beq $t0, $t1, endif
        add $t0,$t0,$t1

endif: xor $t1,$t1,$t1
```

What is the value of address in the compiled machine code?
immediate = 1

If "endif" is below the branch instruction:
immediate = the number of instruction jump over.

If "endif" is above the branch instruction:
immediate = - (2 + the number of instruction jump over)

If jump to the same instruction:
immediate = - 1

Examples

Assembly code

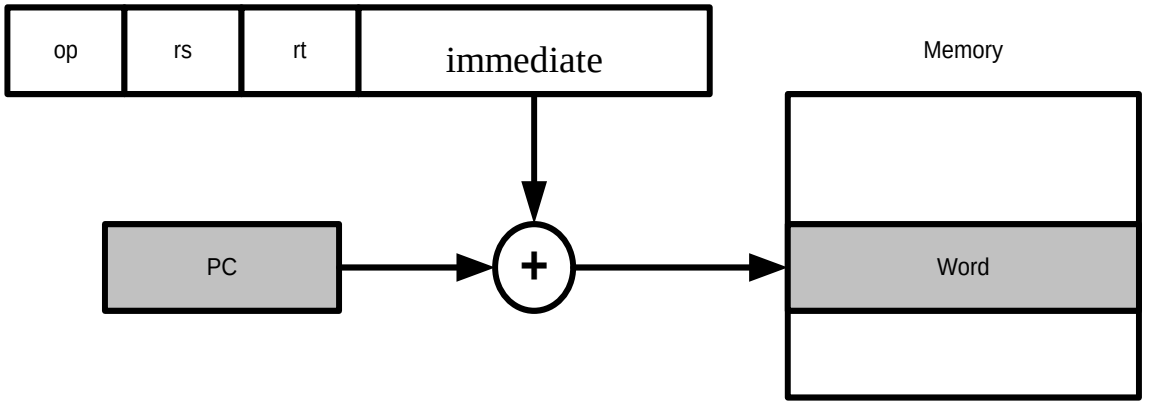
```
        beq $s1, $s2, label
        beq $s1, $s2, label
label:  beq $s1, $s2, label
        beq $s1, $s2, label
        beq $s1, $s2, label
```

op	rs	rt	immediate
0x4	0x11	0x12	0x0001
0x4	0x11	0x12	0x0000
0x4	0x11	0x12	0xFFFF
0x4	0x11	0x12	0xFFFE
0x4	0x11	0x12	0xFFFD

Why?

The way that processor calculates branch address:

compiled machine code:



Address of next inst. (target PC) = PC + 4 + immediate × 4

→ immediate = (target PC – PC)/4 - 1

(target PC – PC)/4-1: number of instructions between current and target instruction.

Examples

Address	Instruction
0x00000004	beq $s1, $s2, label
0x00000008	beq $s1, $s2, label
0x0000000C	beq $s1, $s2, label
0x00000010	beq $s1, $s2, label
0x00000014	beq $s1, $s2, label

label

Address of next inst. (target PC) = PC + 4 + immediate × 4

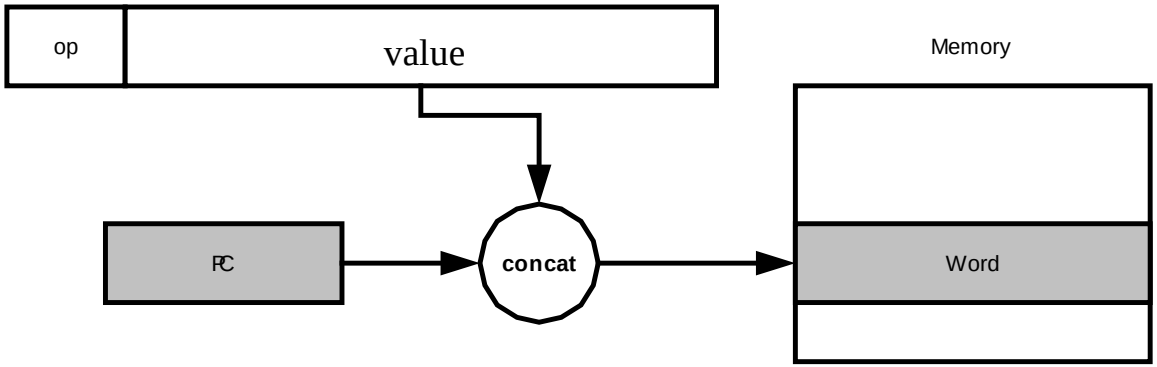
limits the branch distance to -2^{15} to $+2^{15}-1$ (word) instructions from the (instruction after the) branch instruction, but most branches are local anyway.

Jump instruction

assembly code:

```
        j END
        …
END
        …
```

compiled machine code:



Address of next inst. (target PC) = (upper 4 bits of (PC + 4) : value) << 2

Examples

	Address	Instruction
	0x00000004	j lab1
lab2	0x00000008	add $t3, $t4, $t5
	0x0000000C	sub $t5, $t6, $t7
lab1	0x00000010	and $t7, $t8, $t9
	0x00000014	j lab2

Address of next inst. (target PC) = (upper 4 bits of (PC + 4) : 26 bit address) << 2

e.g. What is the machine code of "j lab1" ?

1. find the address of lab1: 0 0 0 0 0 0 0 1 0

2. get the target PC = 0000 00000000000000000000100 00

3. so we get the 26 bit address value 00000000000000000000100

4. j lab1 → 000010 00000000000000000000100

The upper 4 bits of the target PC and (PC+4) MUST be the same. In other words, the process must put the target instruction in a position that having the same upper 4 bits address with the (PC+4).

Summary

- beq label computation
  - Given instruction in machine format, how to compute next pc
  - Given pc and next pc, how to compute immediate
- j label computation
  - Given instruction in machine format, how to compute next pc
  - Given pc and next pc, how to compute offset

Overview from high level to low level

- High-level language program (in C)

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

- Assembly language program (for MIPS)

```
swap:   sll     $t2, $t5, 2
        add     $t2, $t4, $t2
        lw      $s5, 0($t2)
        lw      $s6, 4($t2)
        sw      $s6, 0($t2)
        sw      $s5, 4($t2)
        jr      $ra
```

- Machine (object, binary) code (for MIPS)

```
000000 00000 00101 0001000010000000
000000 00100 00010 0001000000100000
  . . .
```