Chapter Two (2/3)

Machine Language

- **Instructions, like registers and words of data, are also 32 bits long**
  - **Example: `add $t0, $s1, $s2`**
  - **registers have numbers, `$t0=8, $s1=17, $s2=18`**

- **Instruction Format: R-Type**

`000000`	`10001`	`10010`	`01000`	`00000`	`100000`

op	**rs**	**rt**	**rd**	**shamt**	**funct**

op:	opcode
rs:	source register
rt:	target register
rd:	destination register
shamt:	shift amount
funct:	function

Machine Language

- Consider the load-word and store-word instructions,
  - What would the regularity principle have us do?
  - New principle: Good design demands a compromise
- Introduce a new type of instruction format
  - I-type for data transfer instructions
  - other format was R-type for register
- Example: `lw $t0, 32($s2)`

35	18	8	32

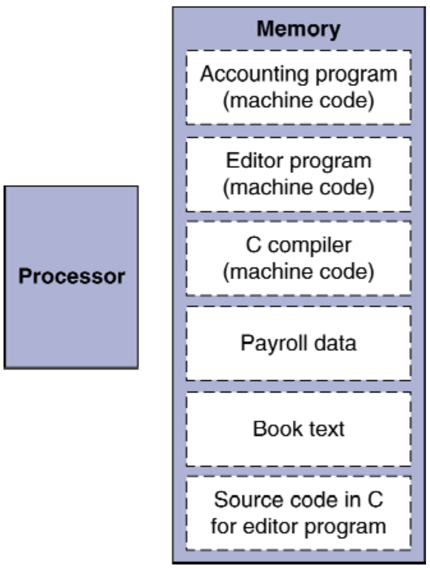
- Instruction Format: I-Type

op	rs	rt	16 bit number

- Where's the compromise? → number of registers

Stored Program Computers

The BIG Picture


Memory
Accounting program (machine code)
Editor program (machine code)
C compiler (machine code)
Payroll data
Book text
Source code in C for editor program
Processor

- Instructions represented in binary, just like data
- Instructions and data stored in memory
- Programs can operate on programs
  - e.g., compilers, linkers, …
- Binary compatibility allows compiled programs to work on different computers
  - Standardized ISAs
- Fetch & Execute Cycle
  - Instructions are fetched and put into a special register
  - Bits in the register "control" the subsequent actions

Control

- **Decision making instructions**
  - **alter the control flow,**
  - **i.e., change the "next" instruction to be executed**
- **MIPS conditional branch instructions:**

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

- **Example:** **if (i==j) h = i + j;**

```
        bne $s0, $s1, Label
        add $s3, $s0, $s1
Label: ....
```

Control

- **MIPS unconditional branch instructions:**

```
j   label
```

op	**26 bit address**

- **Example:**

```
if (i!=j)                        beq $s4, $s5, Lab1
    h=i+j;                       add $s3, $s4, $s5
else                             j   Lab2
    h=i-j;               Lab1:   sub $s3, $s4, $s5
                         Lab2:   ...
```

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

Control

```
C Program
Loop:  g = g + A[i];
       i = i + j;
       if (i != j) goto Loop;

MIPS Assembly
       $s1 = g, $s2 = h, $s3 = i, $s4 = j, $s5 = A[0]

Loop:  add $t1, $s3, $s3   # Temp reg $t1 = 2 * i
       add $t1, $t1, $t1   # Temp reg $t1 = 4 * i
       add $t1, $t1, $s5   # $t1 = address of A[i]
       lw  $t0, 0($t1)     # $t0 = A[i]
       add $s1, $s1, $t0   # g = g + A[i]
       add $s3, $s3, $s4   # i = i + j
       bne $s3, $s4, Loop  # go to Loop if i != j
```

So far:

- <u>Instruction</u> <u>Meaning</u>

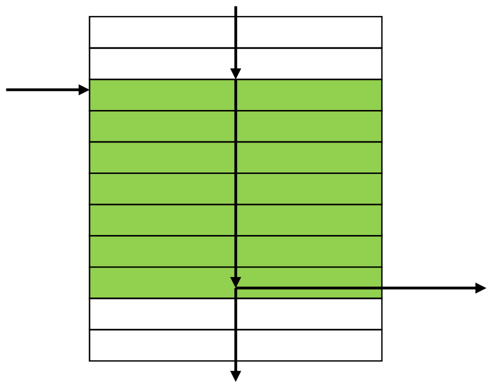
Instruction	Meaning
`add $s1,$s2,$s3`	`$s1 = $s2 + $s3`
`sub $s1,$s2,$s3`	`$s1 = $s2 – $s3`
`lw  $s1,100($s2)`	`$s1 = Memory[$s2+100]`
`sw  $s1,100($s2)`	`Memory[$s2+100] = $s1`
`bne $s4,$s5,L`	Next instr. is at Label if $\$s4 \neq \$s5$
`beq $s4,$s5,L`	Next instr. is at Label if `$s4 = $s5`
`j Label`	Next instr. is at Label

- **Formats:**

R	op	rs	rt	rd	shamt	funct
I	op	rs	rt	16 bit address		
J	op	26 bit address				

Basic Blocks

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



- A compiler identifies basic blocks for optimization
- An advanced processor can accelerate execution of basic blocks

Control Flow

- We have: beq, bne, what about Branch-if-less-than?
- New instruction:

```
                         if  $s1 < $s2 then
                            $t0 = 1
slt $t0, $s1, $s2        else
                            $t0 = 0
```

- Can use this instruction to build `"blt $s1, $s2, Label"`
  — can now build general control structures

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

- Note that the assembler needs a register to do this,
  — there are policy of use conventions for registers

Procedure

- **Purpose**
  - **To structure programs**
    - Easier to understand
    - Allow codes to be reused
  - **To allow programmers to concentrate on just on portion of the task at a time**

- **The execution of a procedure**
  1. **Place parameters in a place where the procedure can access them**
  2. **Transfer control to the procedure**
  3. **Acquire the storage resources needed for the procedure**
  4. **Perform the desired task**
  5. **Place the result value in a place where the calling program can access it**
  6. **Return control to the point of origin**

Use of registers

- **\$a0-\$a3: four arguments**
  \$v0-\$v1: two return values
  \$ra: one return address
- **Procedure call**
  ```
  place the parameters in $a0-$a3
  jal ProcedureAddress
  ```
  - **jumps to** `ProcedureAddress`
  - **saves the return address in \$ra**
- **Return from procedure**
  ```
  places the values in $v0-$v1
  jr $ra
  ```
  - **Jumps to the address stored in \$ra**
- **\$t0-\$t9: 10 temporary registers that are not preserved by the callee**
  - **Caller should save them because any callee is not responsible**
- **\$s0-\$s7: 8 saved registers that must be preserved on a procedure call**
  - **Callee should save them if it wants to use any of them**

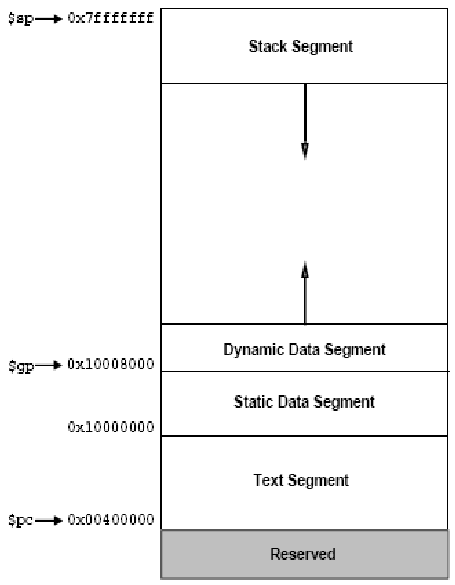
Policy of Use Conventions

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

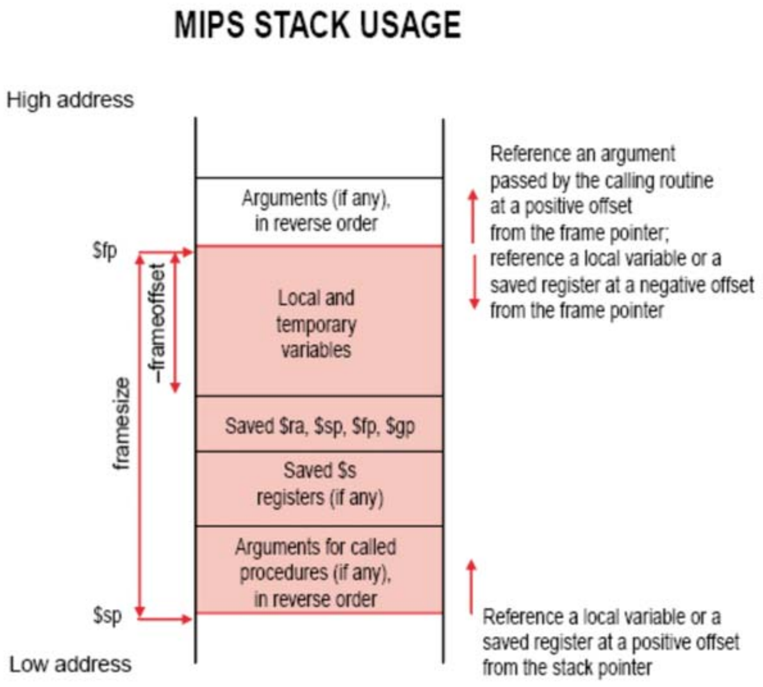
Memory Organization

- **Text: program code**
- **Static data: global variables**
  - **e.g., static variables in C, constant arrays and strings**
  - **$gp initialized to address allowing ±offsets into this segment**
- **Dynamic data: heap**
  - **E.g., malloc in C, new in Java**
- **Stack: automatic storage**

MIPS USER VIRTUAL ADDRESS SPACE



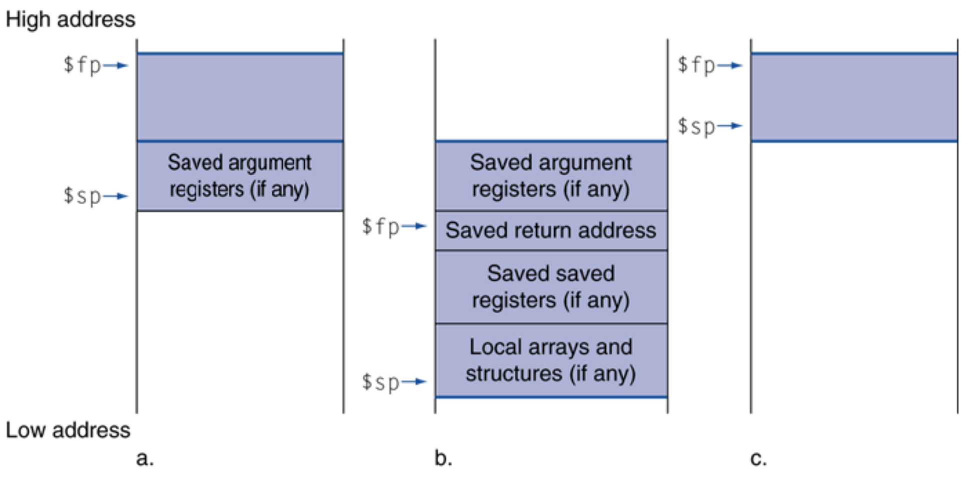
Stack Usage



MIPS STACK USAGE
High address
Arguments (if any), in reverse order
$fp
Local and temporary variables
–frameoffset
framesize
Saved $ra, $sp, $fp, $gp
Saved $s registers (if any)
Arguments for called procedures (if any), in reverse order
$sp
Low address
Reference an argument passed by the calling routine at a positive offset from the frame pointer; reference a local variable or a saved register at a negative offset from the frame pointer
Reference a local variable or a saved register at a positive offset from the stack pointer

Frame Pointer vs. Stack Pointer

- **The segment of the stack containing a procedure's saved registers and local variables is called a procedure frame or activation record.**
- **The frame pointer ($fp) points to the first word of the frame.**
- **The stack pointer ($sp) may change during program execution.**

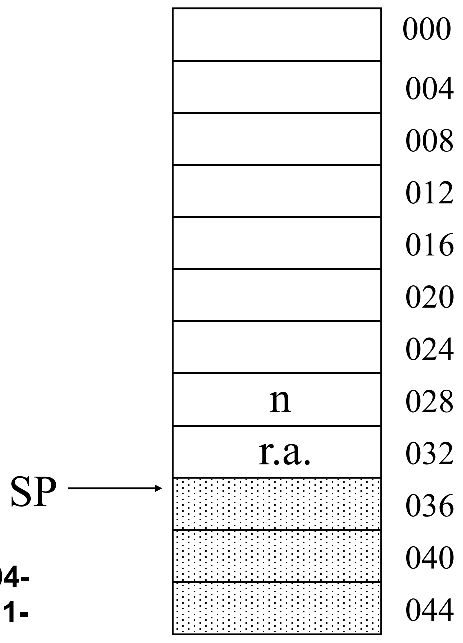


High address
$fp
$sp
Saved argument registers (if any)
Saved argument registers (if any)
Saved return address
Saved saved registers (if any)
Local arrays and structures (if any)
Low address
a.
b.
c.

Nested Procedures

- **The Code starts from address 00 with n=2 and SP=36**

```
00  fact: sub  $sp, $sp, 8    ; push
01        sw   $ra, 4($sp)    ; return addr
02        sw   $a0, 0($sp)    ; argument n

03        slt  $t0, $a0, 1    ; (n < 1)?
04        beq  $t0, $zero, L1 ; if no, go to L1

05        add  $v0, $zero, 1  ; return 1
06        add  $sp, $sp, 8    ; pop
07        jr   $ra            ; return

08  L1:   sub  $a0, $a0, 1    ; n >= 1
09        jal  fact           ; call fact w/(n-1)

10        lw   $a0, 0($sp)    ; return from jal
11        lw   $ra, 4($sp)    ; restore
12        add  $sp, $sp, 8    ; pop

13        mul  $v0, $a0, $v0  ; n * fact(n-1)
14        jr   $ra            ; return
```



Execution:

00(2/28)-01-02-03-04-08(1/28)-09-00(1/20)-01-02-03-04-08(0/20)-09-00(0/12)-01-02-03-04-05-06(0/20)-07-10-11-12(1/28)-13-14-10-11-12(2/36)-13-14

Constants

- **Small constants are used quite frequently (50% of operands)**
  e.g., A = A + 5;
  B = B + 1;
  C = C - 18;
- **Solutions? Why not?**
  - **put 'typical constants' in memory and load them.**
  - **create hard-wired registers (like $zero) for constants like one.**

- **MIPS Instructions:**

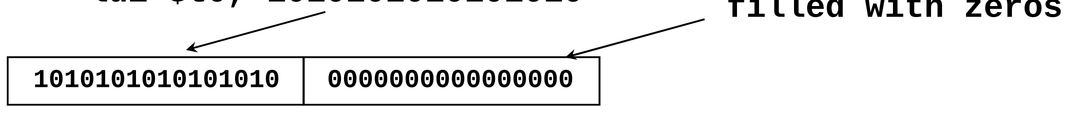
```
addi $r29, $r29, 4
slti $r8,  $r18, 10
andi $r29, $r29, 6
ori  $r29, $r29, 4
```

op	**rs**	**rt**	**immediate**

How about larger constants?

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

```
lui $t0, 1010101010101010
```



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

```
ori $t0, $t0, 1010101010101010
```

	1010101010101010	0000000000000000
ori	0000000000000000	1010101010101010
	1010101010101010	1010101010101010

Addresses in Branches and Jumps

- 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`
`j Label`	Next instruction is at Label

- Formats:

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

- Addresses are not 32 bits
  — How do we handle this with load and store instructions?

Addresses in Branches

- Instructions:

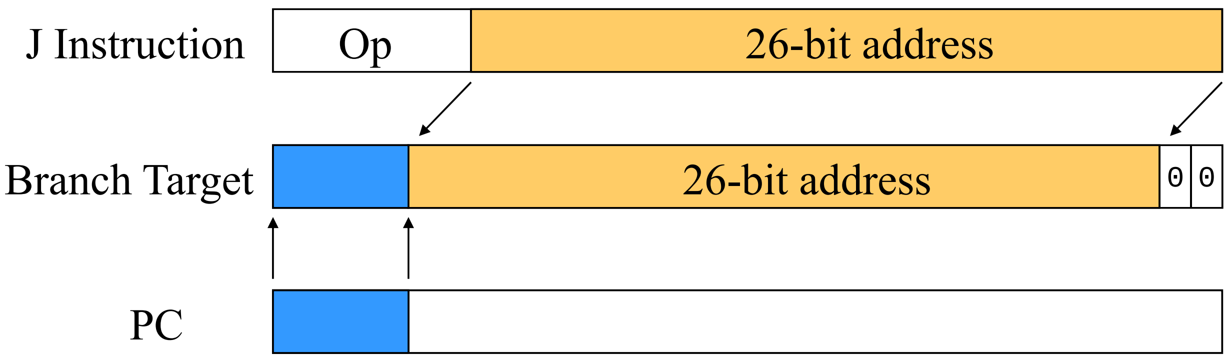
`bne $t4,$t5,Label`	Next instruction is at Label if $\$t4 \neq \$t5$
`beq $t4,$t5,Label`	Next instruction is at Label if $\$t4 = \$t5$

- Formats:

I	op	rs	rt	16 bit address

- Could specify a register (like lw and sw) and add it to address
  - use Instruction Address Register (PC = program counter)
  - most branches are local (principle of locality)
- Jump instructions just use high order bits of PC
  - address boundaries of 256 MB

Additional Material for J instruction



- If branch target is too far to encode with 16-bit offset, assembler rewrites the code
- Example

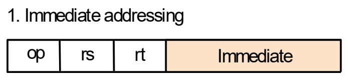
```
        beq $s0,$s1, L1
                        ↓
        bne $s0,$s1, L2
        j L1
    L2: …
```

Assembly Language vs. Machine Language

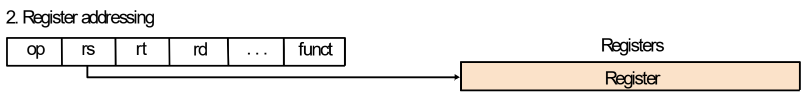
- Assembly provides convenient symbolic representation
  - much easier than writing down numbers
  - e.g., destination first
- Machine language is the underlying reality
  - e.g., destination is no longer first
- Assembly can provide 'pseudo instructions'
  - e.g., “move $t0, $t1” exists only in Assembly
  - would be implemented using “add $t0,$t1,$zero”
- When considering performance you should count real instructions

Addressing Mode (1)

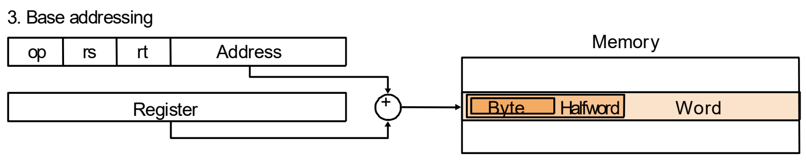
- Immediate addressing
  - The operand isa constant within the instruction itself
  - Example: addi $t0, $t1, 32



- Register addressing
  - The operand is a register
  - Example: add $s0, $s1, $s2



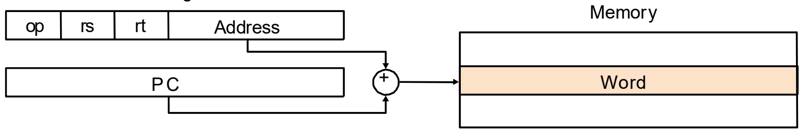
- Base or displacement addressing
  - The operand is at the memory location whose address is the sum of a register and a constant in the instruction
  - Example: lw $s0, 32($s3)



Addressing Mode (2)

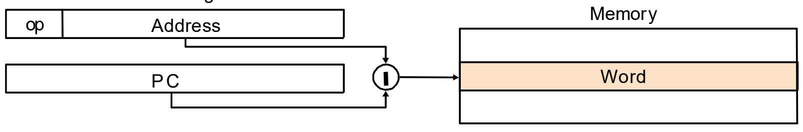
- PC-relative addressing
  - The address is the sum of the PC and a constant in the instruction
  - Example: bne $s0, $s1, Exit

4. PC-relative addressing



- Pseudodirect addressing
  - The jump address is the 26 bits of the instruction concatenated with the upper bits of the PC
  - Example: j Loop

5. Pseudodirect addressing



Overview of MIPS

- simple instructions all 32 bits wide
- very structured, no unnecessary baggage
- only three instruction formats

R	op	rs	rt	rd	shamt	funct
I	op	rs	rt	16 bit address		
J	op	26 bit address				

- rely on compiler to achieve performance
  - what are the compiler's goals?
- help compiler where we can

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
	add	`add $s1, $s2, $s3`	`$s1 = $s2 + $s3`	Three operands; data in registers
Arithmetic	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
	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
Data transfer	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
	branch on equal	`beq $s1, $s2, 25`	if (`$s1 == $s2`) go to PC + 4 + 100	Equal test; PC-relative branch
Conditional	branch on not equal	`bne $s1, $s2, 25`	if (`$s1 != $s2`) go to PC + 4 + 100	Not equal test; PC-relative
branch	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
	jump	`j 2500`	go to 10000	Jump to target address
Uncondi-	jump register	`jr $ra`	go to `$ra`	For switch, procedure return
tional jump	jump and link	`jal 2500`	`$ra` = PC + 4; go to 10000	For procedure call