Mips Hardware

Key Registers: $30, $31, $0

How Mips Executes Programs

Key Thing: PC is incremented before the instruction is executed

Steps to run a program	
1	**Loader** loads program => RAM
2	Additional memory is allocated for the program to use
3	PC = starting address of program in RAM

4	**Fetch-Execute Cycle** runs
	PC = startAddress
	while (true)
	{
	IR = MEM [**PC**]
	PC += 4
	Decode/Execute instruction in **IR**
	}

MIPS Machine Language

Registers = 5 bits

- $d: destination register
- $s: source reg 1
- $t: source reg 2

Immediate value: i

Instruction		**Notes**
jr $s	**PC** = $s	jr $31 - $31 holds the return address - exit the program
lis $d	$d = MEM[PC] PC += 4	Does 2 things:

		1. $d = treats whatever comes after as a 32 bit number, whether it's an instruction or not 2. Skips to the instruction after a. Before executing lis, PC has already been updated to the next instruction
.word i		**The immediate can be:** decimal, hex, Label (PC of label) **directive:** • not an opcode; does not encode a MIPS instruction at all • directive: tells the assembler to encode a 32-bit word at the it's location
mult $s, $t multu $s, $t	**hi:lo** = $s * $t	**hi:** stores upper 32 bits **lo:** stores lower 32 bits
div $s, $t divu $s, $t	$s / $t	**lo** = quotient **hi** = remain sign of the remainder = sign of $s
beq $s, $t, i bne $s, $t, i	if ($s == $t): PC+=i*4 if ($s != $t): PC+=i*4	Since the instruction is already advanced to the next one at the time of execution (i) positive: skip i instructions (from branch) (i) negative: go back |i|-1 instructions (from branch)
slt $d, $s, $t sltu $d, $s, $t	$d = 1 if $s < $t = 0 otherwise	
lw $t i($s) sw $t i($s)	$t = MEM[$s + i] MEM[$s + i] = $t	**Properly Aligned:** address that is a multiple of 4 **Each Address stores 1 byte** **1 word = 4 bytes**
jalr $s	$31 = PC PC = $S	1. $31 = current PC of this instruction (the next instruction) 2. Sets PC to address in $s

Immediate (i)	
branch	decimal, hex, label
word	decimal, hex, label
load/store	decimal, hex

Ranges		
branch i	**16 bits**	$[-2^{15}, 2^{15}-1]$
jr [reg]	**32 bits**	$[-2^{32}, 2^{32}-1]$
.word i	**32 bits**	$[-2^{32}, 2^{32}-1]$

Ex	

add $3, $3, $2
 	
add $2, $2, $1
 	
bne $2, $0, -3
	bne $2, $0, 2
	
	add $3, $3, $2
	
	add $2, $2, $1 (remember, PC is already at the next word)

MEM[$s]	LOAD	STORE
MSB	MEM[$s]	MEM[$s]
LSB	MEM[$s+3]	MEM[$s+3]

***Each address in memory can store 8 bits**

Labels

- assembler directives (not instructions)
- **disappear when:** asm => machine language version

2 Uses	
.word [label]	Assembler converts the label into an **address** • Address of label = instruction at that point
Branch (beq, bne)	Assembler converts the label into an **offset** (immediate value) • Offset = (LabelAddress − PC) / 4

Address of Label

Example	

<pre>0x00 sub $3, $0, $0
      sample:
0x04 add $1, $0, $0
      random: ; is this the end
0x08 mylabel: done: jr $31
      end:
0x0c</pre>
	the assembler will associate:
	1. “sample” → 0x04
	2. “random” & “mylabel” → 0x08
	3. 0x0c → end

Offset Calculation

Example	

<pre>0x00   lis $2
0x04   .word 13
0x08   lis $1
0x0c   .word -1
0x10   add $3, $0, $0
      loop:
0x14   add $3, $3, $2
0x18   add $2, $2, $1
0x1c   bne $2, $0, loop
0x20   jr $31</pre>
	PC = 0x20 (32 in decimal)
	LabelAddress = 0x14 (20 in decimal)
	Offset = (20 - 32) / 4 = -3

Label Rules:

1. unique
2. a label can be defined at the start of any line.
3. can be followed by:
    a. another label
    b. assembly instruction
    c. nothing

Input/Output

Input	Output
lw ← 0xffff0004 1. Reads one byte (8 bits) 2. Store the byte in the destination register (padded with 0s to turn it into a 32-bit word 3. If there are no bytes left to read. -1 is stored	sw → 0xFFFF000C 1. LSB of register ⇒ standard output

Example

```
lis $1
.word 0xFFFF0004
lw $3, 0($1) ; load one character from standard input into $3
```

Example

```
lis $1
.word 0xFFFF000c
lis $2
.word 67      ; ASCII for upper case C
sw $2, 0($1) ; outputs C to standard output.
```

Loops

good to keep loop structures like this

```
loop:  beq i, n, end
        .
        .
        .
        beq $0, $0, loop
```
3

Example: Calculate the value 13+12+11+10.....+1 and store it in $3

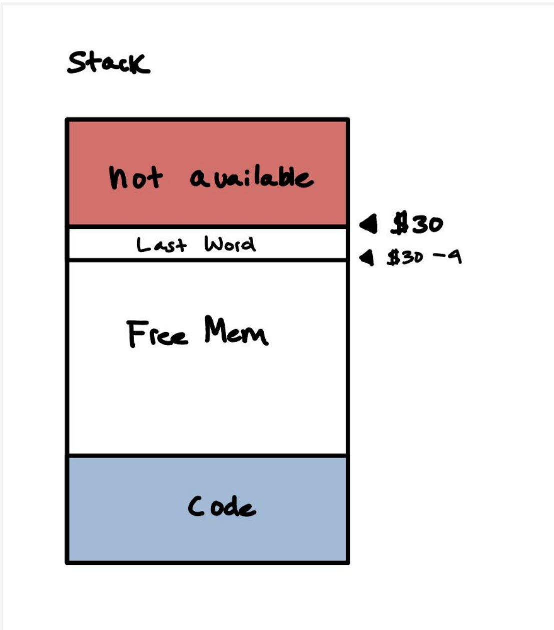
lis $2
.word 13
 	
lis $1
.word -1
 	
add $3, $0, $0
 	
add $3, $3, $2
 	
add $2, $2, $1
 	
bne $2, $0, -3 (remember, PC is already at the next word)
 	
jr $31

Procedures

procedure: label in front of assembly instructions

Stack

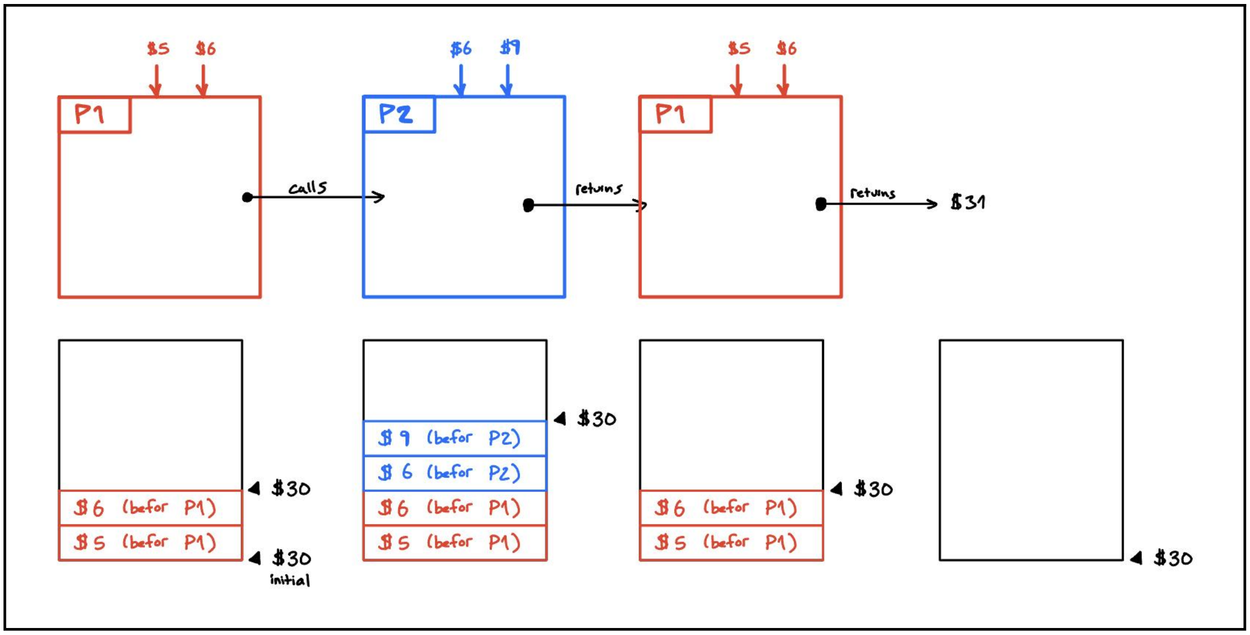
- separates **used** from **unused** memory
- “Increment stack pointer” ⇒ **lower** memory address



3 Things a Procedure should maintain

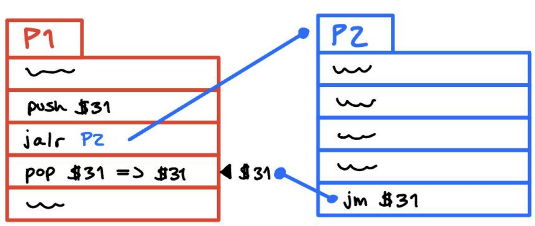
1. Procedure stores/restores parameters
2. procedure stores/restores local variables
3. If the procedure will call another procedure ⇒ store/restore $31

Ex



Push			Pop	
1	**sw**	-4 from $30	**lw**	from $30
2	**Update $30**	subtract 4	**Update $30**	add $4

Call & Return

Calling a Procedure	
	**preserve $31:** (push to stack) ***jalr*** procedure [PROCEDURE RUNS] **restore $31:** (pop from stack)

Ex	
sw $31, -4($30) sub $30, $30, (4)	**preserve $31**

lis $1 .word *procedure1* jalr $1	call the procedure (1) $31 = PC, (2) PC = procedure1
procedure1: . . . jr $31	Procedure runs
lw $31, 0($30) add $30, $30, (4)	**restore $31**

Recursive Procedures

We have to do nothing special if a procedure calls another procedure. As the call chain gets deeper, the stack will grow.

Ex: Recursive Factorial

```
; input: n (non-negative integer) => $1
; output: n! => $3
; base case: 0! = 1
; recursive case: n! = n * (n-1)!

factorial:

   ; base case
   ;................................................
   lis $3
   .word 1

   beq $1, $0, end

   ;................................................
   ; recursive case
   ;................................................
   sub $1, $1, $11     ; n = n-1

   lis $3
   .word factorial

   jalr $3

   add $1, $1, $11     ; n = n + 1
   mult $1, $3
```

```
    mflo $3
  ;.............................................

end:
  jr $31
```

Ex: Write a procedure to sum numbers 1 to N and store result in $3

```
; Sum1ToN adds all numbers 1 to N
; Registers:
;    $1 scratch   (original value should be preserved)
;    $2 input number, N (original value should be preserved)
;    $3 output (don't preserve original value)
Sum1ToN:
sw $1, -4($30)     ; save previous value of $1
sw $2, -8($30)     ; save previous value of $2
lis $1
.word 8
sub $30, $30, $1   ; update stack pointer

add $3, $0, $0     ; initialize to 0
lis $1
.word -1

loop: add $3, $3, $2
add $2, $2, $1
bne $2, $0, loop

lis $1
.word 8
add $30, $30, $1   ; update stack pointer
lw $1, -4($30)     ; restore original value of $1
lw $2, -8($30)     ; restore original value of $2
jr $31  ; return to caller
```

MIPS Assembler

Input	Tokens
Output	Binary

2 Steps	
1	Encode
2	Output

1. Encoding

1. Convert Each **Token** to binary
2. Shift Each binary into position
3. **bitwise(|)** and/or **mask** the result

EXAMPLE 1

Tokens: add $3, $2, $4

1. **Convert Each Token to binary**

Token	DEC	BINARY							
op	0	0000	0000	0000	0000	0000	0000	0000	0000
s	2	0000	0000	0000	0000	0000	0000	0000	0010
t	4	0000	0000	0000	0000	0000	0000	0000	0100
d	3	0000	0000	0000	0000	0000	0000	0000	0011
func	32	0000	0000	0000	0000	0000	0000	0010	0000

2. **Shift Into Position**

Value	Shift	Binary							
op:	(000000) << 26	0000	0000	0000	0000	0000	0000	0000	0000
s:	(00010) << 21	0000	0000	0100	0000	0000	0000	0000	0000
t:	(00100) << 16	0000	0000	0000	0100	0000	0000	0000	0000
d:	(00011) << 11	0000	0000	0000	0000	0001	1000	0000	0000
func:	100000 << 0	0000	0000	0000	0000	0000	0000	0010	0000

3. **Bitwise OR**

```
int instr = (0 << 26) | (2 << 21) | (4 << 16) | (3 << 11) | 32;
```

Result	0000	0000	0100	0100	0001	1000	0010	0000

EXAMPLE 2

Tokens: beq $1, $2, -3

1. **Convert to Binary**

Token	DEC	BINARY							
op	4	0001	0000	0000	0000	0000	0000	0000	0000
s	1		0000	0010	0000	0000	0000	0000	0000
t	2	0000	0000	0000	0010	0000	0000	0000	0000
i	-3	1111	1111	1111	1111	**1111**	**1111**	**1111**	**1101**

2. Shift into Position

Token	Shift	BINARY							
op	(4) << 26	0001	0000	0000	0000	0000	0000	0000	0000
s	(1) << 21		0000	0010	0000	0000	0000	0000	0000
t	(2) << 16	0000	0000	0000	0010	0000	0000	0000	0000
i	-3	1111	1111	1111	1111	**1111**	**1111**	**1111**	**1101**

3. Bitwise Shift + Mask

a. **Mask**

```
  1111111111111111 1111111111111101
& 0000000000000000 1111111111111111 (0xFFFF in hexadecimal)
  ---------------------------------
  0000000000000000 1111111111111101
```

b. **Bitwise OR**

```
int instr = (4 << 26) | (1 << 21) | (2 << 16) | (-3 & 0xFFFF);
```

2. Output

Note:

1. **cout** outputs 1 byte at a time
2. If the number is not a byte, the LSB is outputted

Output terminals interpret the values sent to them as ASCII	
int x = 65 cout << x	unsigned char x = 65 cout << x
65	A
cout assumes we want an integer to be displayed as an integer Each digit: 1. Converted to ASCII 2. sent	cout already assumes a char value is already assumed to be in ASCII. Bits are sent to standard output as is.

Note: if we store 241 (binary 11110001) in the unsigned char?

- While the value is in the numeric range for an unsigned char, it is not a valid ASCII character code
- It will not print anything readable. And that's fine.

Steps:

1. Output 1 byte at a time (MSB → LSB)

```
unsigned char c


c = instr >> 24
cout << c
c = instr >> 16
cout << c
```

```
c = instr >> 8
cout << c
c = instr
cout << c
```

2 Lessons:

1. Our use of unsigned char has nothing to do with “characters” and is just a way to output raw binary data in C++
2. When you view the binary data on your terminal, it will not be human-readable. Most parts of binary-encoded MIPS instructions correspond to non-printable or invalid ASCII characters!