Load Immediate and Skip

MIPS Reference Sheet: `lis $d`

Binary: `0000 0000 0000 0000 dddd d000 0001 0100`

> Instead of specifying a memory address to load from, `lis` loads the next word in memory into the destination register and then skips to the word after that.

Example:

```
lis $7
.word 0x7    ; Lucky number 7
```

To execute `lis $7`, the `.word 0x7` at the current PC is loaded.

Then, PC $\leftarrow$ PC $+$ 4 to perform the skip.

Example:

Write a program that adds 27 to 42 and stores the sum in $3.

Example:

Write a program that adds 27 to 42 and stores the sum in $3.

```
lis $5              ; load immediate and skip $5 <- 27
.word 27
lis $6              ; load immediate and skip $6 <- 42
.word 42
add $3, $5, $6      ; $3 <- $5 + $6
jr $31              ; PC <- $31 jump to address in $31
```

When asked to “Write a program ...", you should **return**, even if not explicitly asked to do so; i.e. your program should terminate properly.

Consider the following program:

Address	Assembly	Hexadecimal
0x0000	`lis $1`	`0x0000 0814`
0x0004	`lis $2`	`0x0000 1014`
0x0008	`jr $0`	`0x0000 0008`
0x000c	`jr $31`	`0x03e0 0008`

What value (in decimal) is loaded into register $1?

What value (in decimal) is loaded into register $2?

What does the program do?

Branching

Two options: Branch on Equal and Branch on Not Equal

Compares contents of two registers; if true, branch; i.e. modify PC by the given (immediate) offset number of words.

MIPS Reference Sheet: `beq $s, $t, i` and `bne $s, $t, i`

Binary: `0001 00ss ssst tttt iiii iiii iiii iiii`

- `i` is an integer offset (unit is number of words)
- $\text{PC} \leftarrow \text{PC} + \texttt{i} \times 4$

Recall: PC stores address of next instruction.

What does `beq $0, $0, -1` do?

Set Less Than

Two forms: Set Less Than and Set Less Than Unsigned

Compares contents of two registers (as either two's complement or unsigned numbers); sets destination register with result.

MIPS Reference Sheet: `slt $d, $s, $t` and `sltu $d, $s, $t`

Binary: `0000 00ss ssst tttt dddd d000 0010 1010`

- Sets `$d` $\leftarrow$ 1 if `$s < $t`; otherwise `$d` $\leftarrow$ 0
- CS 241 does **not** have Set Greater Than, Set Equal To, etc.
- With branching, we can implement conditionals, looping, etc.

Conditional example: Write a program to compute the absolute value of `$1` and store result in `$1`.

Conditional example: Write a program to compute the absolute value of `$1` and store result in `$1`.

```
slt $2, $1, $0    ; compare $1 < 0, is $1 negative?
beq $2, $0, 1     ; if $1 positive skip next instr
sub $1, $0, $1    ; negate $1: $1 <- 0-$1
jr $31            ; return
```

Alternative:

```
slt $2, $0, $1
bne $2, $0, 1
sub $1, $0, $1
jr $31
```

Loop example: Write a program that adds all the even numbers from 2 to 20 (inclusive) and stores the sum in register $3.

Loop example: Write a program that adds all the even numbers from 2 to 20 (inclusive) and stores the sum in register $3.

```
lis $1
.word 20
add $3, $0, $0
add $3, $3, $1
lis $2
.word 2
sub $1, $1, $2
bne $1, $0, -5  ; loop to add $3, $3, $1
jr $31
```

Multiplication and Division

MIPS Reference Sheet: `mult $s, $t` and `div $s, $t`

Where is the destination register?

How many bits are needed for the product of two 32-bit numbers?

Hint: consider multiplying (in decimal) 1000×1000.

Multiplication and Division

MIPS Reference Sheet: `mult $s, $t` and `div $s, $t`

Where is the destination register?

How many bits are needed for the product of two 32-bit numbers?

Hint: consider multiplying (in decimal) 1000×1000.

- Product could require 64 bits - too big for a single register
- Product stored in special registers: `hi:lo` $\leftarrow$ `$s*$t`
- Division has a quotient (stored in `lo`) and remainder (`hi`)
- Also, unsigned versions: `multu` and `divu`

Accessing `hi` and `lo`

MIPS Reference Sheet: `mfhi $d` and `mflo $d`

> Move from `hi` (or `lo`) simply copies the contents of `hi` (or `lo`) to a destination register.

Example: Given `$1` stores the base address of an array and `$2` stores a valid index, write a program that loads the value into `$3`.

Accessing `hi` and `lo`

MIPS Reference Sheet: `mfhi $d` and `mflo $d`

> Move from `hi` (or `lo`) simply copies the contents of `hi` (or `lo`) to a destination register.

Example: Given `$1` stores the base address of an array and `$2` stores a valid index, write a program that loads the value into `$3`.

```
lis $4
.word 4
mult $2, $4
mflo $4
add $4, $1, $4
lw $3, 0($4)
jr $31
```

Example:

Write a program that checks if `$2` evenly divides `$1`.

If true, `$3` $\leftarrow$ 1; otherwise `$3` $\leftarrow$ 0.

Registers `$1` and `$2` must remain unchanged.

```
div $1, $2
mfhi $3
bne $3, $0, ??? ; if remainder != 0 branch
```

Where do we branch to?

- Maybe we should write the rest of the code and fill this in later.

```
div $1, $2
mfhi $3
bne $3, $0, 4   ; if remainder != 0 branch
lis $4          ; case 1: remainder == 0
.word 1         ; set $3 <- 1
add $3, $4, $0
beq $0, $0, 1
add $3, $0, $0  ; case 2: set $3 <- 0
jr $31
```

Assembly Language

Assembly language replaces the binary encoding of machine language instructions with easier to use mnemonics; i.e. its more English-like code.

- Readability, less chance of errors, etc
- Can make an Assembler to automatically translate ASM to ML
- 1 line of assembly translates to 1 line of machine code
- Has extra features to simplify coding (directives: e.g. `.word`)
- Allows for comments and extra whitespace (stripped out at pre-processing)

Assembly Language Labels

Assemblers allow programmers to label instructions and to use the labels within the assembly language instruction so programmers do not have to manually calculate jump addresses or branch offsets.

Format: `label:  operation operands`

Example: replace `i` in `beq` instruction:

```
; ABS program
slt $2, $1, $0
beq $2, $0, 1
sub $1, $0, $1
jr $31
```

```
; Label in beq instr
     slt $2, $1, $0
     beq $2, $0, foo
     sub $1, $0, $1
foo: jr $31
```

Revisiting Loops

Loop example: Write a program that adds all the even numbers from 2 to 20 (inclusive) and stores the sum in register $3.

```
lis $1
.word 20
add $3, $0, $0
add $3, $3, $1
lis $2
.word 2
sub $1, $1, $2
bne $1, $0, -5  ; loop to add $3, $3, $1
jr $31
```

Revisiting Loops

Loop example: Write a program that adds all the even numbers from 2 to 20 (inclusive) and stores the sum in register `$3`.

```
lis $1
.word 20
add $3, $0, $0
add $3, $3, $1
lis $2
.word 2
sub $1, $1, $2
bne $1, $0, -5  ; loop to add $3, $3, $1
jr $31
```

Why load value 2 at each iteration of the loop?

Revisiting Loops

Modifying code might invalidate offsets. As a programmer, we don't want to manually update offsets and addresses, etc.

Let the assembler do the work for us!

```
lis $1
.word 20
lis $2          ; move this out of loop
.word 2
add $3, $0, $0
add $3, $3, $1
sub $1, $1, $2
bne $1, $0, -5 ; This okay?
jr $31
```

```
lis $1
.word 20
lis $2
.word 2
add $3, $0, $0
add $3, $3, $1
sub $1, $1, $2
bne $1, $0, -3
jr $31
```

```
     lis $1
     .word 20
     lis $2
     .word 2
     add $3, $0, $0
top:
     add $3, $3, $1
     sub $1, $1, $2
     bne $1, $0, top
     jr $31
```

`top` is assigned memory adddress `0x14`

Assembler computes: $(\texttt{top} - \text{PC})/4$ = $(\texttt{0x14} - \texttt{0x20})/4$ = -3 in `bne`

Assigning Memory Addresses to Labels

Remember, whitespace, comments and labels are for programmers to more easily read, write and organize code. They do not get translated in machine code!

When assigning a memory address to a line label:

- Blank lines are simply stripped out.
- Whitespace after labels is removed.
- Only instructions (and `.word`) are assigned addresses.
- A label is assigned the memory address of the instruction that follows it.

- A label may appear at the end of your code and will be assigned the memory address of the word after your program.