Lab_Offsets

CMPUT 229

University of Alberta

Outline

Lab_Offsets: Decoding Branch Instructions

- In this lab you will learn to decode a MIPS Branch instruction.

Branches

Branches in MIPS are control structures (i.e., if-then structures in higher level programming)

Instruction	What does it do?
`bgez` \$s, offset	Jumps to PC $+$ offset $\times$ 4 if \$s ≥ 0.
`bgezal` \$s, offset	Sets \$ra $=$ PC, jumps to PC $+$ offset $\times$ 4 if \$s ≥ 0.
`bltz` \$s, offset	Jumps to PC $+$ offset $\times$ 4 if \$s < 0.
`bltzal` \$s, offset	Sets \$ra $=$ PC, jumps to PC $+$ offset $\times$ 4 if \$s < 0.
`beq` \$s, \$t, offset	Jumps to PC $+$ offset $\times$ 4 if \$s $=$ \$t.
`bne` \$s, \$t, offset	Jumps to PC $+$ offset $\times$ 4 if \$s $\neq$ \$t.
`blez` \$s, offset	Jumps to PC $+$ offset $\times$ 4 if \$s ≤ 0.
`bgtz` \$s, offset	Jumps to PC $+$ offset $\times$ 4 if \$s > 0.

Branch Offset

- Then branch **offset** indicates where to jump.
- If a branch is taken, the PC is set to $PC + offset \times 4$.
- Offset: how many words up or down we need to move.
  - Offset $\in [-2^{15}, 2^{15} - 1]$
- Encoding:

31 30 29 28 27 26	25 24 23 22 21	20 19 18 17 16	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
OpCode	Register s	Register t	Branch Offset (16-bit Immediate)

Branch Offset: An Example

- Suppose there is a program containing a branch-not-equals, `bne`.
- Suppose $\$s4 \neq \$t2$.

	......
0x80001000	`add $t0, $s3, $0`
0x80001004	`srl $t1, $t0, 4`
0x80001008	`lw $t2, 8($t1)`
0x8000100c	`bne $s4, $t2, Done`
0x80001010	`addi $t2, $t2, 32`
0x80001014	`sb $s4, 0($t2)`
0x80001018	`Done: jr $ra`

Branch Offset: An Example

	......
0x80001000	`add   $t0, $s3, $0`
0x80001004	`srl   $t1, $t0, 4`
0x80001008	`lw    $t2, 8($t1)`
0x8000100c	**`bne   $s4, $t2, Done`** ← Being executed
0x80001010	`addi  $t2, $t2, 32` ← PC = 0x80001010
0x80001014	`sb    $s4, 0($t2)`
0x80001018	`Done: jr    $ra`

Branch Offset: An Example

Address	Instruction
	......
0x80001000	add $t0, $s3, $0
0x80001004	srl $t1, $t0, 4
0x80001008	lw $t2, 8($t1)
0x8000100c	**bne $s4, $t2, Done** ← Being executed
0x80001010	addi $t2, $t2, 32 ← PC = 0x80001010
0x80001014	sb $s4, 0($t2)
0x80001018	Done: jr $ra

OpCode	Reg	Reg	Offset
5	20	10	**2**

Branch Offset: An Example

	......
0x80001000	add $t0, $s3, $0
0x80001004	srl $t1, $t0, 4
0x80001008	lw $t2, 8($t1)
0x8000100c	**bne $s4, $t2, Done** ← Being executed
0x80001010	addi $t2, $t2, 32
0x80001014	sb $s4, 0($t2)
0x80001018	**Done: jr $ra** ← PC (to be executed)

PC = 0x80001010 + 2 * 4 = 0x80001018

Hexadecimal: Why?

- Hexadecimal is used for easy conversion to and from binary.
- It is easier to read and represent HEX than binary.
- Also: conversion between hexadecimal and ASCII is easy.

Simple Concepts of Hex and ASCII

- All data in a computer is represented in binary (i.e., only 0s and 1s).
- Hexadecimal: one way to read/print groups of bits rather than individual bits. Each hex value represents 4 binary digits.
- American Standard Code for Information Interchange (ASCII): a representation standard for characters and symbols.
- `.asciiz` predirectives in the data segment are null-terminated strings (similar to C strings). A "string" in MIPS must have a null character at the end.

Translation between ASCII and Hex

- Hex to ASCII

Input:	0x4
Output:	'4'

INPUT	ASCII	Binary
$(00000100)_2$	0x34	$(00110100)_2$

```
0000 0100 OR 0011 0000 = 0011 0100
```

Translation between ASCII and Hex

- ASCII to Hex

Input:	'4'
Output:	0x4

ASCII	Binary	OUTPUT
0x34	$(00110100)_2$	$(00000100)_2$

```
0011 0100 AND 0000 1111 = 0000 0100
```

Subroutines

- *Subroutines* in MIPS are calls to another portion of the code.
- **Why?** ← Just like "procedures" or "functions" in higher level languages.
- `jal` is an instruction to make the jump to a symbol (marker for a piece of code).
- `jr` returns control to the caller.
- Conventions: $a registers for arguments, $v for return values.
- Conventions will be examined in more in detail in upcoming labs.

Masking Operations

- The smallest segment of memory that can be loaded/stored in main memory is *one byte*.
- There is no instruction in MIPS to set/get individual bits inside a word.
- Masks allow us to read and set individual bits and perform some arithmetic functions.

Masking Operations - AND

AND

1. When using `ones` in the mask, the corresponding bits are preserved or queried.
2. When using `zeros` in the mask, the corresponding bits are turned off.

Example 1: Querying a bit.

`Mask:`	0000 1000
`Number:`	1111 0101
`Result:`	0000 **0**000

Example 2: Extracting information.

`Mask:`	0000 1111
`Number:`	1010 1000
`Result:`	**0000** *1000*

Masking Operations - OR

OR

1. When using `ones` in the mask, the corresponding bits are set to one.
2. When using `zeros` in the mask, the corresponding bits are preserved.

Example 1: Set to one.

`Mask:`	0000 1111
`Number:`	1101 0101
`Result:`	1101 **1111**

Example 2: Preserving bits.

`Mask:`	0000 1111
`Number:`	1010 0000
`Result:`	**1010** *1111*

Masking Operations - XOR

OR

1. When using `ones` in the mask, the corresponding bits are inverted (toggled).
2. When using `zeros` in the mask, the corresponding bits are preserved.

`Mask:`	0000 1111
`Number:`	1101 0101
`Result:`	1101 **1010**

Shifting and Rotating Bits

- Shifting is basically moving bits from one location to another.
- Shifting can also be viewed as an arithmetic operation
  - Each time you shift to the **left** a bit you multiply **times 2**.
  - Each time you shift to the **right** a bit you divide **by 2**.
- *Rotating* is the same as shifting, but bits get carried back to the opposite side.

Shift:	4_{10} left logical
Number:	1101 0101
Result:	0101 0000

Shift:	4_{10} right logical
Number:	1101 0101
Result:	0000 1101

Shift:	4_{10} right arithmetic
Number:	1101 0101
Result:	1111 1101

Shift:	4_{10} rotate left
Number:	1101 0101
Result:	0101 1101

Shifting as an Arithmetic Operation

- In any positional `base-b` numeral system, adding zeros to the right implies multiplying times the base (power of the radix).
  - $\mathbf{1040_{10} * 10_{10} = 10400_{10}}$ (sll 1).
  - $\mathbf{0011_2 * 10_2 = 0110_2}$, (sll 1) = ($3_{10} * 2_{10} = 6_{10}$).
  - $\mathbf{001F_{16} * 10_{16} = 01F0_{16}}$, (sll 1) ($31_{10} * 16_{10} = 496_{10}$).
- Analogously, shifting to the right implies a division by the base.
  - $\mathbf{0110_2 / 10_2 = 0011_2}$, (srl 1) ($6_{10} / 2_{10} = 3_{10}$).
  - Caveat: for signed numbers in two's complement you need an **arithmetic shift** instead of a **logical shift**.
  - $\mathbf{1011_2 / 10_2 = 1101_{16}}$, (sra 1) ($-5_{10} / 2_{10} \neq -3_{10}$).

Lab 2 Assignment

You need to create a **subroutine** called `disassembleBranch` that:

- Takes an argument $a0 $\rightarrow$ the address of a branch instruction.
- Translates from a binary representation to text.
- Output:
  - If instruction is not a branch $\rightarrow$ no output is generated.
  - If instruction is a MIPS branch, $\rightarrow$ the instruction must be printed in the screen.
- Please refer to the lab specification for all details on the output.

Assignment Tips

- Read specifications very carefully. Pay special attention to what you have to include - we don't want a `main` method.
- Test your assignments on the lab machines before you submit. That's where we'll be marking them.
- Use the test cases provided to debug your code.
- Look at the marksheet to get an idea of how the grading will be done.
- Style marks are easy marks. Format your code like the `example.s` file we provided, and write good comments.
- Be sure to submit code that runs and loads; otherwise, you will lose marks.

Questions?