CSE 31
Computer Organization

Lecture 12 – MIPS: Logical Operators
Machine Instruction Format (1)

Announcement

- Lab #6 this week
  - Due next week
- HW #3 (from zyBooks)
  - Due Monday (10/8)
- Project #1
  - Due Monday (10/22)
    - Don’t start late, you won’t have time!
- Reading assignment
  - Chapter 4.1-4.9 of zyBooks (Reading Assignment #4)
    - Make sure to do the Participation Activities
    - Due Monday (10/15)

Review

- Functions called with `jal`, return with `jr $ra`.
- The stack is your friend: Use it to save anything you need. Just leave it the way you found it!
- Instructions we know so far…

  Arithmetic: `add, addi, sub, addu, addiu, subu`
  Memory: `lw, sw, lb, sb, lbu`
  Decision: `beq, bne, slt, slti, sltu, sltiu`
  Unconditional Branches (Jumps): `j, jal, jr`
- Registers we know so far
  - All of them!
  - There are CONVENTIONS when calling procedures!

Quiz

```
r: ...      # R/W $s0,$v0,$t0,$a0,$sp,$ra,mem
   ...      ### PUSH REGISTER(S) TO STACK?
   jal e    # Call e
   ...      # R/W $s0,$v0,$t0,$a0,$sp,$ra,mem
   jr $ra   # Return to caller of r

e: ...      # R/W $s0,$v0,$t0,$a0,$sp,$ra,mem
   jr $ra   # Return to r
```

What does `r` have to push on the stack before "`jal e`"?

a) 1 of ($s0,$sp,$v0,$t0,$a0,$ra)
b) 2 of ($s0,$sp,$v0,$t0,$a0,$ra)
c) 3 of ($s0,$sp,$v0,$t0,$a0,$ra)
d) 4 of ($s0,$sp,$v0,$t0,$a0,$ra)
e) 5 of ($s0,$sp,$v0,$t0,$a0,$ra)

Quiz

```
r: ...      # R/W $s0,$v0,$t0,$a0,$sp,$ra,mem
   ...      ### PUSH REGISTER(S) TO STACK?
   jal e    # Call e
   ...      # R/W $s0,$v0,$t0,$a0,$sp,$ra,mem
   jr $ra   # Return to caller of r

e: ...      # R/W $s0,$v0,$t0,$a0,$sp,$ra,mem
   jr $ra   # Return to r
```

What does `r` have to push on the stack before "`jal e`"?

		Saved	Volatile! -- need to push
a)	1 of	($s0,$sp	$v0,$t0,$a0,$ra)
b)	2 of	($s0,$sp	$v0,$t0,$a0,$ra)
c)	3 of	($s0,$sp	$v0,$t0,$a0,$ra)
☺ **d)**	**4 of**	**($s0,$sp**	**$v0,$t0,$a0,$ra)**
e)	5 of	($s0,$sp	$v0,$t0,$a0,$ra)

Bitwise Operations

- So far, we've done arithmetic (**`add`**, **`sub`**,**`addi`**), mem access (**`lw`** and **`sw`**), & branches and jumps.
- All of these instructions view contents of register as a single quantity (e.g., signed or unsigned int)
- New Perspective: View register as 32 raw bits rather than as a single 32-bit number
  - Since registers are composed of 32 bits, wish to access individual bits (or groups of bits) rather than the whole.
- Introduce two new classes of instructions
  - Logical & Shift Ops

Logical Operators (1/3)

- Two basic logical operators:
  - AND: outputs 1 only if all inputs are 1
  - OR: outputs 1 if at least one input is 1
- Truth Table: standard table listing all possible combinations of inputs and resultant output

A	B	A AND B	A OR B
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

Logical Operators (2/3)

- Logical Instruction Syntax:

  1 2,3,4

  - where

    1) operation name

    2) register that will receive value

    3) first operand (register)

    4) second operand (register) or immediate (numerical constant)

- In general, can define them to accept > 2 inputs, but in the case of MIPS assembly, these accept exactly 2 inputs and produce 1 output
  - Again, rigid syntax, simpler hardware

Logical Operators (3/3)

- Instruction Names:
  - `and`, `or`: Both of these expect the third argument to be a register
  - `andi`, `ori`: Both of these expect the third argument to be an immediate
- MIPS Logical Operators are all bitwise, meaning that bit 0 of the output is produced by the respective bit 0's of the inputs, bit 1 by the bit 1's, etc.
  - C: Bitwise AND is **&** (e.g., `z = x & y;`)
  - C: Bitwise OR is **|** (e.g., `z = x | y;`)

Uses of Logical Operators (1/3)

- Note that **and**ing a bit with **0** produces a **0** at the output while **and**ing a bit with **1** produces the original bit.
- This can be used to create a **mask**.
  - Example:

```
1011 0110 1010 0100 0011 | 1101 1001 1010
0000 0000 0000 0000 0000 | 1111 1111 1111
```

  - The result of **and**ing these:

```
0000 0000 0000 0000 0000 | 1101 1001 1010
```

mask:

mask last 12 bits

Uses of Logical Operators (2/3)

- The second bitstring in the example is called a **mask**. It is used to isolate the rightmost 12 bits of the first bitstring by masking out the rest of the string (e.g. setting to all **0**s).
- Thus, the **`and`** operator can be used to set certain portions of a bitstring to **0**s, while leaving the rest alone.
  - In particular, if the first bitstring in the above example were in **`$t0`**, then the following instruction would mask it:

```
andi   $t0,$t0,0xFFF
```

Uses of Logical Operators (3/3)

- Similarly, note that **or**ing a bit with **1** sets a **1** at the output while **or**ing a bit with **0** keeps the original bit.
- Often used to force certain bits to **1**s.
  - For example, if **$t0** contains **0x12345678**, then after this instruction:

```
ori   $t0, $t0, 0xFFFF
```

    … **$t0** will contain **0x1234FFFF**
    - (i.e., the high-order 16 bits are untouched, while the low-order 16 bits are forced to **1**s).

Shift Instructions (review) (1/4)

- Move (shift) all the bits in a word to the left or right by a number of bits.
  - Example: shift right by 8 bits

  - Example: shift left by 8 bits

Shift Instructions (2/4)

- Shift Instruction Syntax:

  1 2,3,4

  …where

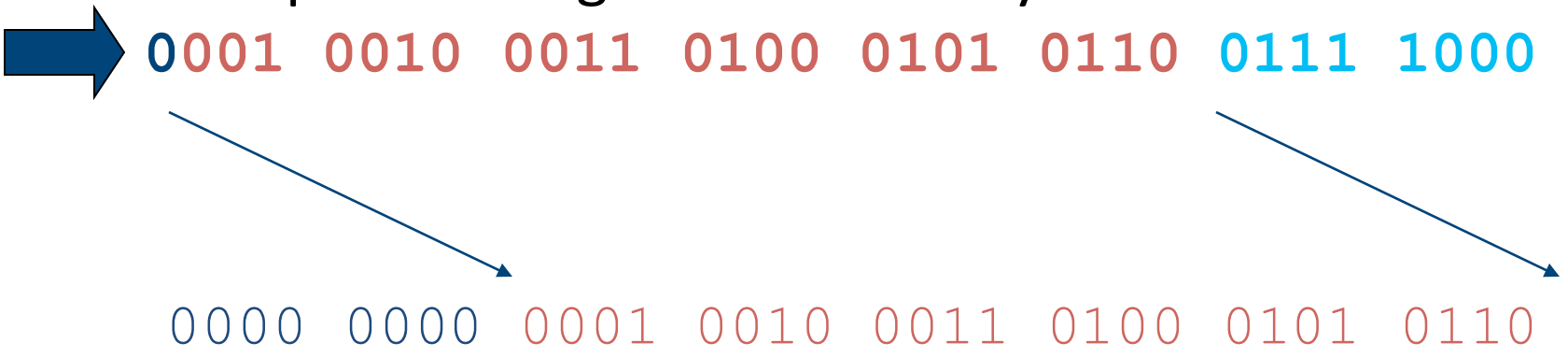
  1) operation name
  2) register that will receive value
  3) first operand (register)
  4) shift amount (constant < 32)

- MIPS shift instructions:

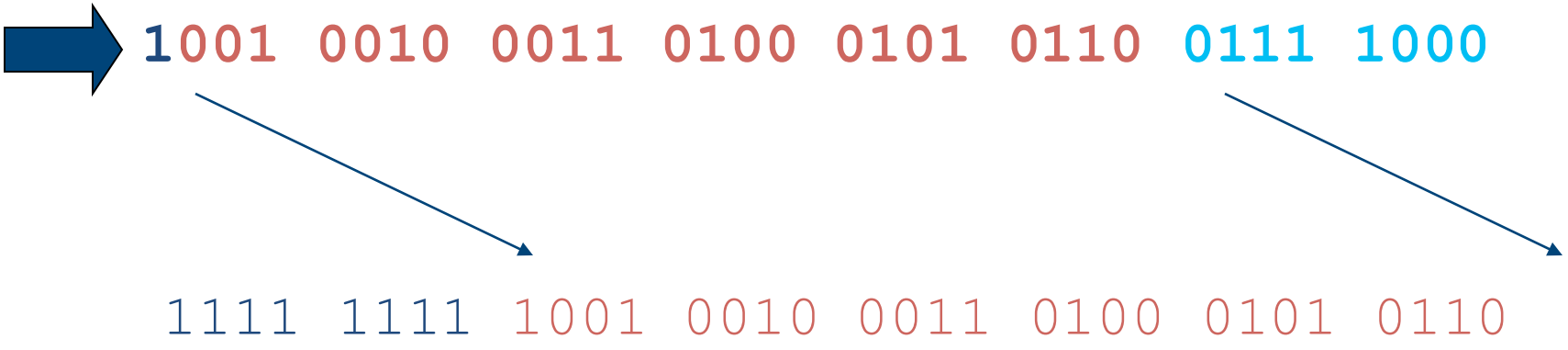
  1. `sll` (shift left logical): shifts left and fills emptied bits with 0s
  2. `srl` (shift right logical): shifts right and fills emptied bits with 0s
  3. `sra` (shift right arithmetic): shifts right and fills emptied bits by sign extending

Shift Instructions (3/4)

- Example: shift right arithmetic by 8 bits



- Example: shift right arithmetic by 8 bits



Shift Instructions (4/4)

- Since shifting is faster than multiplication, a good compiler usually notices when C code multiplies by a power of 2 and compiles it to a shift instruction:

  `a *= 8;` (in C)

  would compile to:

  `sll  $s0,$s0,3` (in MIPS)

- Likewise, shift right to divide by powers of 2 (rounds towards $-\infty$)
  - remember to use `sra`

Summary

- Logical and Shift Instructions
  - Operate on bits individually, unlike arithmetic, which operate on entire word.
  - Use to isolate fields, either by masking or by shifting back and forth.
  - Use shift left logical, **sll**,for multiplication by powers of 2
  - Use shift right logical, **srl**,for division by powers of 2 of unsigned numbers (**unsigned int**)
  - Use shift right arithmetic, **sra**,for division by powers of 2 of signed numbers (**int**)
- New Instructions:

  and, andi, or, ori, sll, srl, sra
- That's all you need to know about MIPS!

Levels of Representation (abstractions)

High Level Language Program (e.g., C)

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

Compiler

Assembly Language Program (e.g.,MIPS)

```
lw   $t0, 0($s2)
lw   $t1, 4($s2)
sw   $t1, 0($s2)
sw   $t0, 4($s2)
```

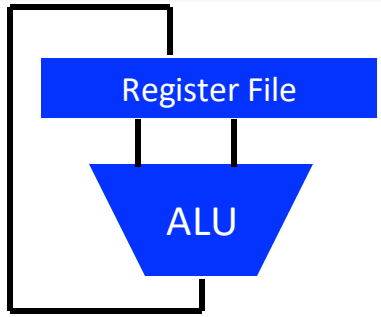
Assembler

Machine Language Program (MIPS)

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

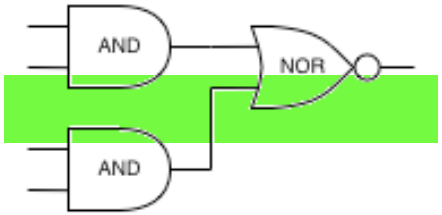
Machine Interpretation

Hardware Architecture Description (e.g., block diagrams)



Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)



Overview – Instruction Representation

- Big idea: stored program
  - consequences of stored program
- Instructions as numbers
- Instruction encoding
- MIPS instruction format for arithmetic instructions
- MIPS instruction format for Immediate, Data transfer instructions

Big Idea: Stored-Program Concept

- Where are programs stored when they are being run?
  - How are they stored in memory?
- Computers built on 2 key principles:
  - Instructions are represented as bit patterns - can think of these as numbers.
  - Therefore, entire programs can be stored in memory to be read or written just like data.
- Simplifies SW/HW of computer systems:
  - Memory technology for data also used for programs

Consequence #1: Everything Addressed

- Since all instructions and data are stored in memory, everything has a memory address: instructions, data words
  - both branches and jumps use these
- C pointers are just memory addresses: they can point to anything in memory
  - Unconstrained use of addresses can lead to nasty bugs; up to you in C; limits in Java
- One register keeps address of instruction being executed: "Program Counter" (PC)
  - Basically a pointer to memory: Intel calls it Instruction Address Pointer, a better name

Consequence #2: Binary Compatibility

- Programs are distributed in binary form
  - Programs bound to specific instruction set
  - Different version for Macintoshes and PCs
- New machines want to run old programs ("binaries") as well as programs compiled to new instructions
  - Leads to "backward compatible" instruction set evolving over time
  - Selection of Intel 8086 in 1981 for 1st IBM PC is major reason latest PCs still use 80x86 instruction set (Pentium 4); could still run program from 1981 PC today

Instructions as Numbers (1/2)

- Currently all data we work with is in words (32-bit blocks):
  - Each register is a word.
  - **`lw`** and **`sw`** both access memory one word at a time.
- So how do we represent instructions?
  - Remember: Computer only understands 1s and 0s, so "`add $t0,$0,$0`" is meaningless.
  - MIPS wants simplicity: since data is in words, make instructions into words too!

Instructions as Numbers (2/2)

- One word is 32 bits, so divide instruction word into "fields".
- Each field tells processor something about the instruction.
- We could define different fields for each instruction, but MIPS is based on simplicity, so define 3 basic types of instruction formats:
  - I-format
  - J-format
  - R-format
  - What do these letters (I, J, R) stand for?

Instruction Formats

- I-format: used for instructions with immediates, `lw` and `sw` (since offset counts as an immediate), and branches (`beq` and `bne`),
  - (but not the shift instructions; later)
- J-format: used for `j` and `jal`
- R-format: used for all other instructions
- Why 3 different formats?
  - It will soon become clear why the instructions have been partitioned in this way.

R-Format Instructions (1/5)

- Define "fields" of the following number of bits each: 6 + 5 + 5 + 5 + 5 + 6 = 32

6	5	5	5	5	6

- For simplicity, each field has a name:

opcode	rs	rt	rd	shamt	funct

- Important: On these slides and in book, each field is viewed as a 5- or 6-bit unsigned integer, not as part of a 32-bit integer.
  - Consequence: 5-bit fields can represent any number 0-31, while 6-bit fields can represent any number 0-63.

R-Format Instructions (2/5)

- What do these field integer values tell us?
  - `opcode`: partially specifies what instruction it is
    - Note: This number is equal to `0` for all R-Format instructions.
  - `funct`: combined with `opcode`, this number exactly specifies the instruction
- Question: Why aren't `opcode` and `funct` a single 12-bit field?
  - We'll answer this later.

R-Format Instructions (3/5)

- More fields:
  - `rs` (Source Register): generally used to specify register containing first operand
  - `rt` (Target Register): generally used to specify register containing second operand (note that name is misleading)
  - `rd` (Destination Register): generally used to specify register which will receive result of computation

R-Format Instructions (4/5)

- Notes about register fields:
  - Each register field is exactly 5 bits, which means that it can specify any unsigned integer in the range 0-31. Each of these fields specifies one of the 32 registers by number.
  - The word "generally" was used because there are exceptions
    - **`mult`** and **`div`** have nothing important in the **`rd`** field since the dest registers are **`hi`** and **`lo`**
    - **`mfhi`** and **`mflo`** have nothing important in the **`rs`** and **`rt`** fields since the source is determined by the instruction.

R-Format Instructions (5/5)

- Final field:
  - `shamt`: This field contains the amount a shift instruction will shift by. Shifting a 32-bit word by more than 31 is useless, so this field is only 5 bits (so it can represent the numbers 0-31).
  - This field is set to `0` in all but the shift instructions.
- For a detailed description of field usage for each instruction, see MIPS_Reference_Sheet.pdf in CatCourses (You can bring with you to all exams)

R-Format Example (1/2)

- MIPS Instruction:

  `add $t0,$t1,$t2`

  `add $8,$9,$10`

  `opcode` = 0 (look up in table)

  `funct` = 32 (look up in table)

  `rd` = 8 (destination)

  `rs` = 9 (first operand)

  `rt` = 10 (second operand)

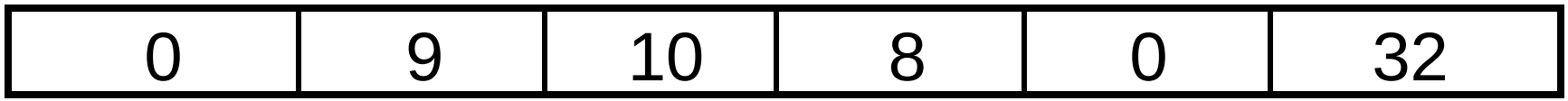
  `shamt` = 0 (not a shift)

R-Format Example (2/2)

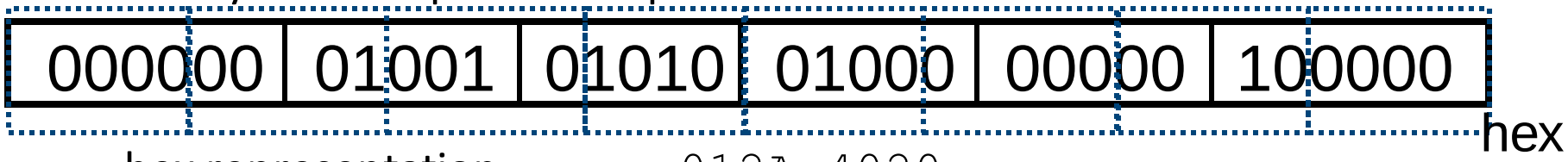
- MIPS Instruction:

  `add $8,$9,$10`

  Decimal number per field representation:



  Binary number per field representation:



  hex representation: $012A\ 4020_{hex}$

  decimal representation: $19,546,144_{ten}$

  Called a Machine Language Instruction

I-Format Instructions (1/4)

- What about instructions with immediates?
  - 5-bit field only represents numbers up to the value 31: immediates may be much larger than this
  - Ideally, MIPS would have only one instruction format (for simplicity): unfortunately, we need to compromise
- Define new instruction format that is partially consistent with R-format:
  - First notice that, if instruction has immediate, then it uses at most 2 registers.

I-Format Instructions (2/4)

- Define "fields" of the following number of bits each: 6 + 5 + 5 + 16 = 32 bits

6	5	5	16

  - Again, each field has a name:

opcode	rs	rt	immediate

  - Key Concept: Only one field is inconsistent with R-format. Most importantly, `opcode` is still in same location.

I-Format Instructions (3/4)

- What do these fields mean?
  - `opcode`: same as before except that, since there's no `funct` field, `opcode` uniquely specifies an instruction in I-format
    - This also answers question of why R-format has two 6-bit fields to identify instruction instead of a single 12-bit field: in order to be consistent as possible with other formats while leaving as much space as possible for immediate field.
  - `rs`: specifies a register operand (if there is one)
  - `rt`: specifies register which will receive result of computation (this is why it's called the target register "`rt`") or other operand for some instructions.

I-Format Instructions (4/4)

- The Immediate Field:
  - **`addi`**, **`slti`**, **`sltiu`**, the immediate is sign-extended to 32 bits. Thus, it's treated as a signed integer.
  - 16 bits ➔ can be used to represent immediate up to 2^{16} different values
  - This is large enough to handle the offset in a typical **`lw`** or **`sw`**, plus a vast majority of values that will be used in the **`slti`** instruction.
  - We'll see what to do when the number is too big later

I-Format Example (1/2)

- MIPS Instruction:

  `addi $s5,$s6,-50`

  `addi $21,$22,-50`

  `opcode` = 8 (look up in table in book)

  `rs` = 22 (register containing operand)

  `rt` = 21 (target register)

  `immediate` = -50 (by default, this is decimal)

I-Format Example (2/2)

- MIPS Instruction:

```
addi    $21,$22,-50
```

Decimal/field representation:

8	22	21	-50

Binary/field representation:

001000	10110	10101	1111111111001110

hexadecimal representation: 22D5 FFCE$_{hex}$

decimal representation: 584,449,998$_{ten}$

Quiz

Which instruction has same representation as 35_{ten}?

a) add $0, $0, $0
b) subu $s0,$s0,$s0
c) lw $0, 0($0)
d) addi $0, $0, 35
e) subu $0, $0, $0

opcode	rs	rt	rd	shamt	funct
opcode	rs	rt	rd	shamt	funct
opcode	rs	rt	offset		
opcode	rs	rt	immediate		
opcode	rs	rt	rd	shamt	funct

Registers numbers and names:
0: $0, .. 8: $t0, 9: $t1, ..15: $t7, 16: $s0, 17: $s1, .. 23: $s7

Opcodes and function fields (if necessary)

`add`: opcode = 0, funct = 32
`subu`: opcode = 0, funct = 35
`addi`: opcode = 8
`lw`: opcode = 35

Quiz

Which instruction has same representation as 35_{ten}?

a) add $0, $0, $0
b) subu $s0,$s0,$s0
c) lw $0, 0($0)
d) addi $0, $0, 35
e) subu $0, $0, $0

opcode	rs	rt	rd	shamt	funct
opcode	rs	rt	rd	shamt	funct
opcode	rs	rt	offset		
opcode	rs	rt	immediate		
opcode	rs	rt	rd	shamt	funct

Registers numbers and names:
0: $0, .. 8: $t0, 9: $t1, ..15: $t7, 16: $s0, 17: $s1, .. 23: $s7

Opcodes and function fields (if necessary)

`add`: opcode = 0, funct = 32
`subu`: opcode = 0, funct = 35
`addi`: opcode = 8
`lw`: opcode = 35