CS 61C:
Great Ideas in Computer Architecture
MIPS Instruction Formats

Instructors:

Vladimir Stojanovic and Nicholas Weaver

http://inst.eecs.Berkeley.edu/~cs61c/sp16

Levels of Representation/Interpretation

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

Compiler

Assembly Language Program (e.g., MIPS)

Assembler

Machine Language Program (MIPS)

Machine Interpretation

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

Architecture Implementation

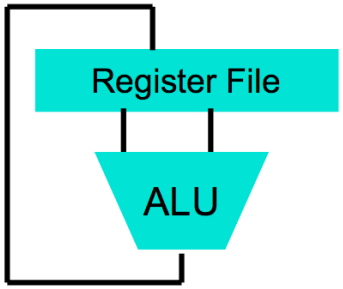
Logic Circuit Description (Circuit Schematic Diagrams)

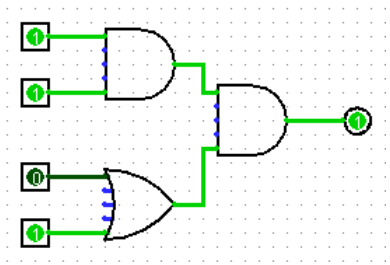
```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw   $t0, 0($2)
lw   $t1, 4($2)
sw   $t1, 0($2)
sw   $t0, 4($2)
```

Anything can be represented as a *number*, i.e., data or instructions

```
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
```





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
- 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 Example (1/2)

- MIPS Instruction:

  `add  $8,$9,$10`

  `opcode` = 0 (look up in table in book)

  `funct` = 32 (look up in table in book)

  `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:

0	9	10	8	0	32

  Binary number per field representation:

000000	01001	01010	01000	00000	100000

hex

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

  Called a Machine Language Instruction

opcode	rs	rt	rd	shamt	funct

I-Format Instructions (1/2)

- 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 (2/2)

- 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.
  - Later, we'll see what to do when a value is too big for 16 bits

I-Format Example (1/2)

- MIPS Instruction:

```
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 in assembly code)

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 $FFCE_{hex}$

Clicker/Peer Instruction

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

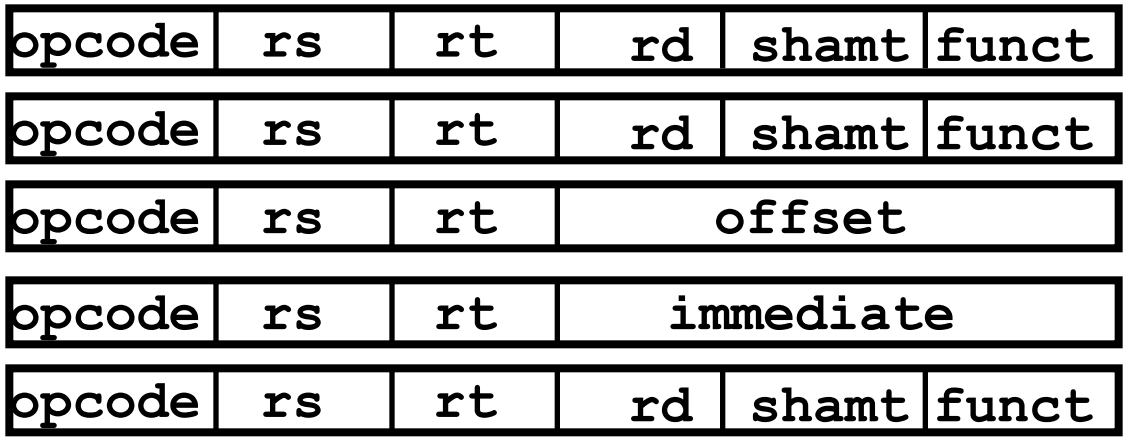
a) add $0, $0, $0

b) subu $s0,$s0,$s0

c) lw $0, 0($0)

d) addi $0, $0, 35

e) subu $0, $0, $0



Registers numbers and names:

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

Opcodes and function fields:

`add`: opcode = 0, funct = 32

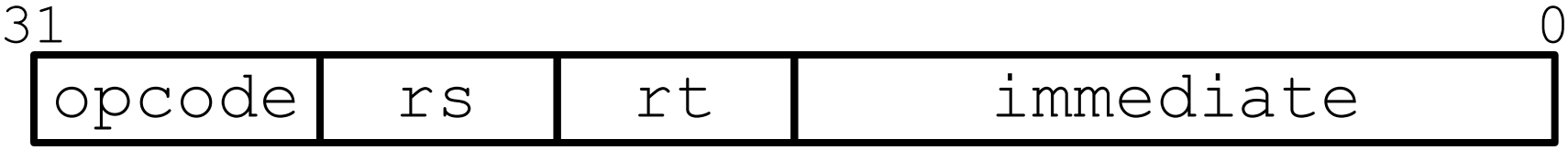
`subu`: opcode = 0, funct = 35

`addi`: opcode = 8

`lw`: opcode = 35

Branching Instructions

- `beq` and `bne`
  - Need to specify a target address if branch taken
  - Also specify two registers to compare
- Use I-Format:



  - `opcode` specifies `beq` (4) vs. `bne` (5)
  - `rs` and `rt` specify registers
  - How to best use `immediate` to specify addresses?

Branching Instruction Usage

- Branches typically used for loops (`if-else`, `while`, `for`)
  - Loops are generally small (< 50 instructions)
  - Function calls and unconditional jumps handled with jump instructions (J-Format)
- **Recall:** Instructions stored in a localized area of memory (Code/Text)
  - Largest branch distance limited by size of code
  - Address of current instruction stored in the program counter (PC)

PC-Relative Addressing

- PC-Relative Addressing: Use the `immediate` field as a two's complement offset to PC
  - Branches generally change the PC by a small amount
  - Can specify ± 2^{15} addresses from the PC

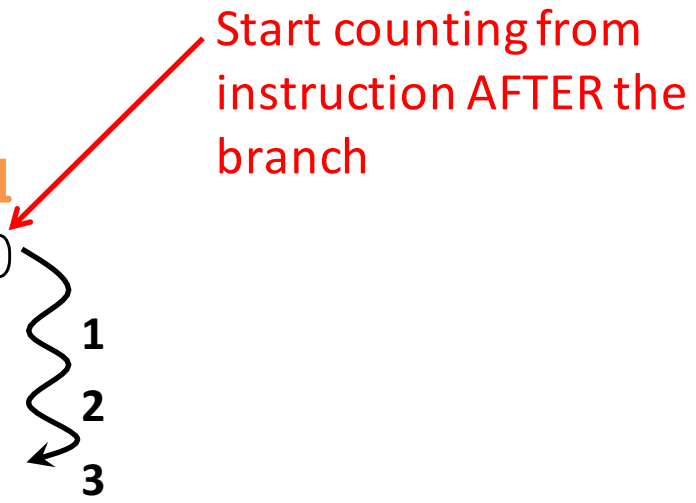
Branch Calculation

- If we don't take the branch:
  - `PC = PC + 4` = next instruction
- If we do take the branch:
  - `PC = (PC+4) + (immediate*4)`
- **Observations:**
  - `immediate` is number of instructions to jump (remember, specifies words) either forward (+) or backwards (–)
  - Branch from `PC+4` for hardware reasons; will be clear why later in the course

Branch Example (1/2)

- MIPS Code:

```
Loop: beq   $9,$0,End
      addu  $8,$8,$10
      addiu $9,$9,-1
      j     Loop
End:
```



- I-Format fields:

  opcode = 4 (look up on Green Sheet)

  rs = 9 (first operand)

  rt = 0 (second operand)

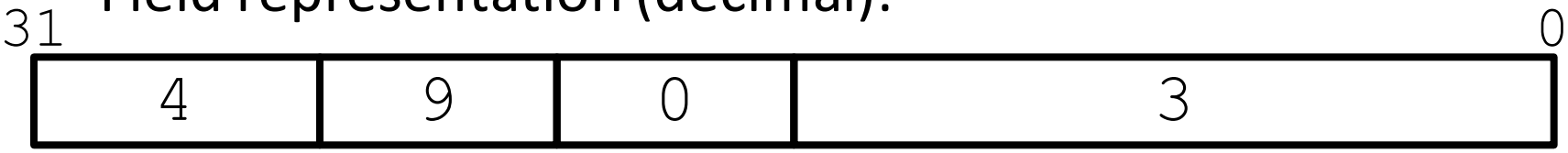
  immediate = 3

Branch Example (2/2)

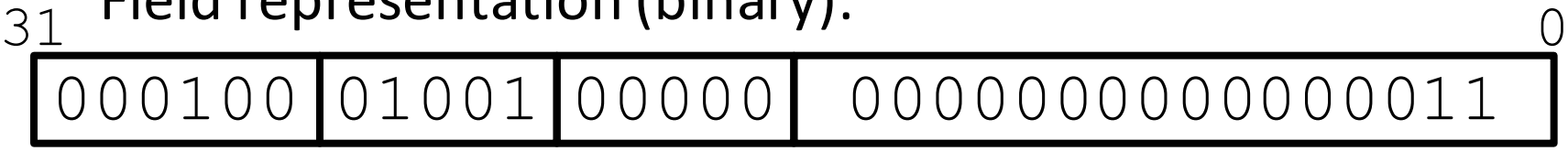
- MIPS Code:

```
Loop: beq   $9,$0,End
      addu  $8,$8,$10
      addiu $9,$9,-1
      j     Loop
End:
```

Field representation (decimal):



Field representation (binary):



Questions on PC-addressing

- Does the value in branch immediate field change if we move the code?
  - If moving individual lines of code, then yes
  - If moving all of code, then no
- What do we do if destination is > 2^{15} instructions away from branch?
  - Other instructions save us
  -
```
beq $s0,$0,far          bne $s0,$0,next
# next instr      →     j   far
                  next: # next instr
```

J-Format Instructions (1/4)

- For branches, we assumed that we won't want to branch too far, so we can specify a *change* in the PC
- For general jumps (`j` and `jal`), we may jump to *anywhere* in memory
  - Ideally, we would specify a 32-bit memory address to jump to
  - Unfortunately, we can't fit both a 6-bit `opcode` and a 32-bit address into a single 32-bit word

J-Format Instructions (2/4)

- Define two "fields" of these bit widths:

- As usual, each field has a name:

- **Key Concepts:**
  - Keep `opcode` field identical to R-Format and I-Format for consistency
  - Collapse all other fields to make room for large target address

J-Format Instructions (3/4)

- We can specify 2^{26} addresses
  - Still going to word-aligned instructions, so add `0b00` as last two bits (multiply by 4)
  - This brings us to 28 bits of a 32-bit address
- Take the 4 highest order bits from the PC
  - Cannot reach *everywhere*, but adequate almost all of the time, since programs aren't that long
  - Only problematic if code straddles a 256MB boundary
- If necessary, use 2 jumps or `jr` (R-Format) instead

J-Format Instructions (4/4)

- Jump instruction:
  - New PC = { (PC+4)[31..28], target address, 00 }
- Notes:
  - { , , } means concatenation
    { 4 bits , 26 bits , 2 bits } = 32 bit address
    - Book uses || instead
  - Array indexing: [31..28] means highest 4 bits
  - For hardware reasons, use PC+4 instead of PC

MAL vs. TAL

- True Assembly Language (TAL)
  - The instructions a computer understands and executes
- MIPS Assembly Language (MAL)
  - Instructions the assembly programmer can use (includes pseudo-instructions)
  - Each MAL instruction becomes 1 or more TAL instruction

Assembler Pseudo-Instructions

- Certain C statements are implemented unintuitively in MIPS
  - e.g. assignment (`a=b`) via add $zero
- MIPS has a set of "pseudo-instructions" to make programming easier
  - More intuitive to read, but get translated into actual instructions later
- Example:

```
move dst,src
```

translated into

```
add dst,src,$zero
```

Assembler Pseudo-Instructions

- List of pseudo-instructions:
  http://en.wikipedia.org/wiki/MIPS_architecture#Pseudo_instructions
  - List also includes instruction translation
- Load Address (`la`)
  - `la dst,label`
  - Loads address of specified label into `dst`
- Load Immediate (`li`)
  - `li dst,imm`
  - Loads 32-bit immediate into `dst`
- MARS has additional pseudo-instructions
  - See Help (F1) for full list

Assembler Register

- Problem:
  - When breaking up a pseudo-instruction, the assembler may need to use an extra register
  - If it uses a regular register, it'll overwrite whatever the program has put into it
- Solution:
  - Reserve a register ($1 or $at for "assembler temporary") that assembler will use to break up pseudo-instructions
  - Since the assembler may use this at any time, it's not safe to code with it

Dealing With Large Immediates

- How do we deal with 32-bit immediates?
  - Sometimes want to use immediates > ± 2^{15} with `addi`, `lw`, `sw` and `slti`
  - Bitwise logic operations with 32-bit immediates
- **Solution:** Don't mess with instruction formats, just add a new instruction
- Load Upper Immediate (`lui`)
  - `lui reg,imm`
  - Moves 16-bit `imm` into upper half (bits 16-31) of `reg` and zeros the lower half (bits 0-15)

`lui` Example

- Want: `addiu $t0,$t0,0xABABCDCD`
  - This is a pseudo-instruction!
- Translates into:

```
 lui   $at,0xABAB     # upper 16
ori   $at,$at,0xCDCD  # lower 16
addu  $t0,$t0,$at     # move
```

- Now we can handle everything with a 16-bit immediate!

Clicker Question

Which of the following place the address of LOOP in $v0?

```
1) la $t1, LOOP
   lw $v0, 0($t1)

2) jal LOOP
   LOOP: addu $v0, $ra, $zero

3) la $v0, LOOP
```

	1	2	3
A)	T,	T,	T
B)	T,	T,	F
C)	F,	T,	T
D)	F,	T,	F
E)	F,	F,	T

Administrivia

- Project 2-1 is out – start early
- HW1 (ungraded) C-to-MIPS practice problems (due 02/14 @ 23:59:59)
  - Will help you a lot with the midterm so don't skip
- Piazza Etiquette
  - Please don't post code. We do not debug over piazza. Come to OH instead!
  - Search through other posts, FAQs before posting a question

Integer Multiplication (1/3)

- Paper and pencil example (unsigned):

```
  Multiplicand   1000   8
  Multiplier    x1001   9
                 1000
                0000
               0000
             +1000
             --------
             01001000   72
```

- m bits x n bits = m + n bit product

Integer Multiplication (2/3)

- In MIPS, we multiply registers, so:
  - 32-bit value x 32-bit value = 64-bit value
- Syntax of Multiplication (signed):
  - `mult` register1, register2
  - Multiplies 32-bit values in those registers & puts 64-bit product in special result registers:
    - puts product upper half in hi, lower half in lo
  - hi and lo are 2 registers separate from the 32 general purpose registers
  - Use `mfhi` register & `mflo` register to move from hi, lo to another register

Integer Multiplication (3/3)

- Example:
  - in C: `a = b * c;`
  - in MIPS:
    - let `b` be `$s2`; let `c` be `$s3`; and let a be `$s0` and `$s1` (since it may be up to 64 bits)

```
mult $s2,$s3    # b*c
mfhi $s0        # upper half of
                # product into $s0
mflo $s1        # lower half of
                # product into $s1
```

- Note: Often, we only care about the lower half of the product
  - Pseudo-inst. `mul` expands to `mult/mflo`

Integer Division (1/2)

- Paper and pencil example (unsigned):

```
                    1001   Quotient
Divisor 1000 | 1001010    Dividend
             -1000
                 10
                 101
                 1010
                -1000
                   10 Remainder
                (or Modulo result)
```

- Dividend = Quotient x Divisor + Remainder

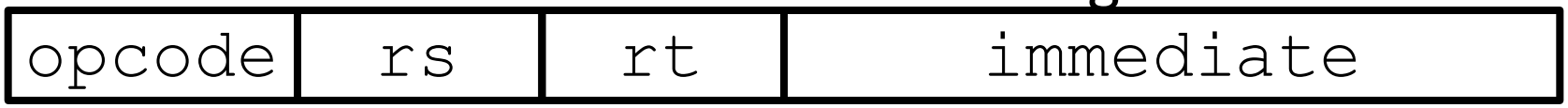
Integer Division (2/2)

- Syntax of Division (signed):
  - `div` register1, register2
  - Divides 32-bit register 1 by 32-bit register 2:
  - puts remainder of division in `hi`, quotient in `lo`
- Implements C division (`/`) and modulo (`%`)
- Example in C: `a = c / d;` `b = c % d;`
- in MIPS: `a↔$s0;` `b↔$s1;` `c↔$s2;` `d↔$s3`

```
div  $s2,$s3  # lo=c/d, hi=c%d
mflo $s0      # get quotient
mfhi $s1      # get remainder
```

Summary

- I-Format: instructions with immediates, `lw`/`sw` (offset is immediate), and `beq`/`bne`
  - But not the shift instructions
  - Branches use PC-relative addressing



- J-Format: `j` and `jal` (but not `jr`)
  - Jumps use absolute addressing

- R-Format: all other instructions