CprE 381: Computer Organization and Assembly Level Programming

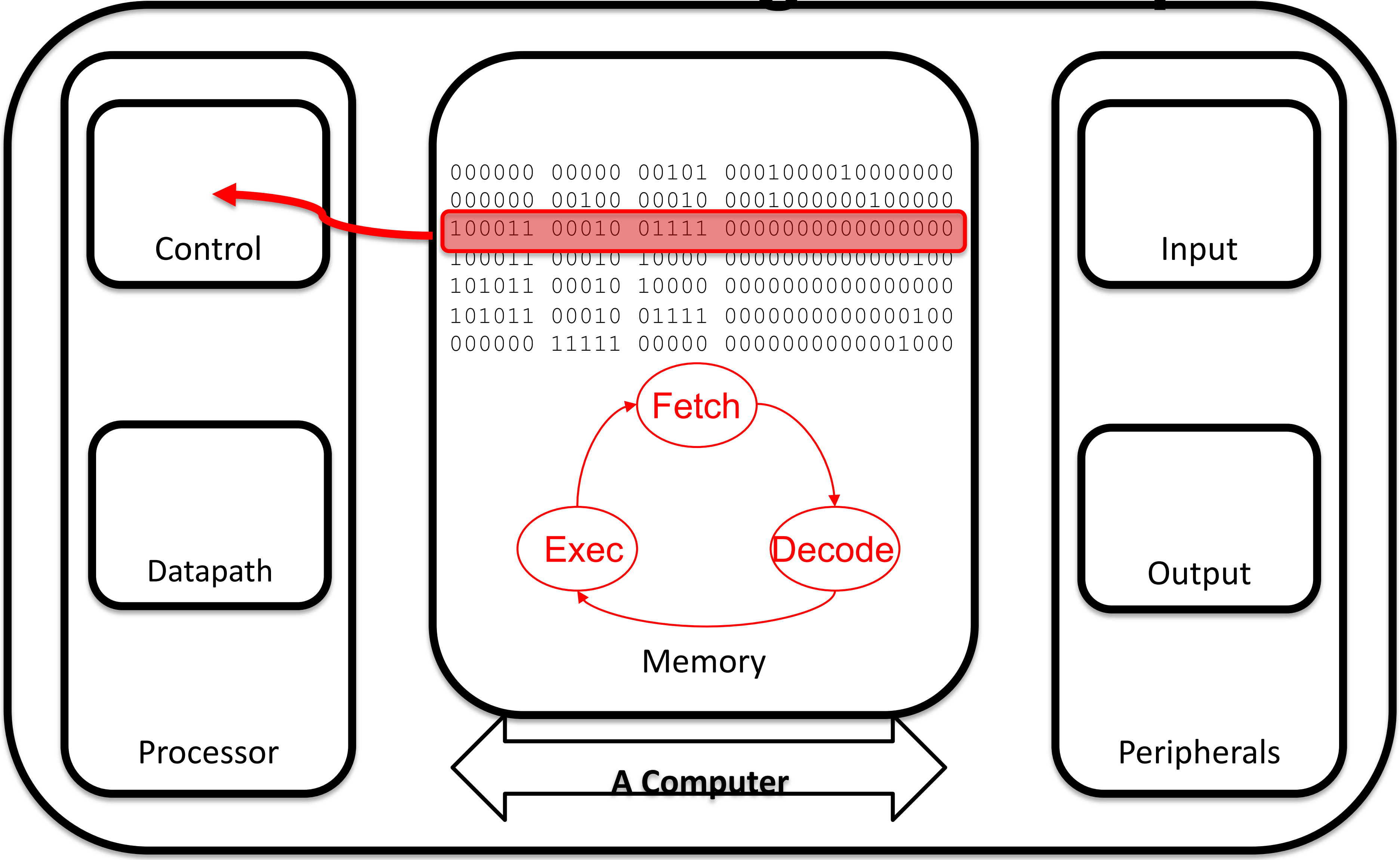
MIPS Machine Code

Henry Duwe
Electrical and Computer Engineering
Iowa State University

Administrative

- Lab 4 this week
  - Final individual lab
  - Term Project starts next week – make sure you have your team in Canvas by end of your lab session
- Feedback: your peers find VHDL and MIPS interesting

Review: Stored Program Computer



```
000000 00000 00101 0001000010000000
000000 00100 00010 0001000000100000
100011 00010 01111 0000000000000000
100011 00010 10000 0000000000000100
101011 00010 10000 0000000000000000
101011 00010 01111 0000000000000100
000000 11111 00000 0000000000001000
```

Review: MIPS Instruction Encoding

- MIPS Instructions encoded in three different formats, depending upon the operands
  - R-format, I-format, J-format
- MIPS instruction formats are all 32 bits in length
  - Regularity is simpler and improves performance
- A 6 bit opcode indicates format and general function
- See MIPS “green card” for more complete info
  - pdf on book companion material website-- http://booksite.elsevier.com/9780124077263/index.php

Review: R – Format

- Uses three registers: one for destination and two for source
- Used by ALU instructions

	6	5	5	5	5	6
Instruction Bits	**OP=0**	**rs**	**rt**	**rd**	**shamt**	**funct**
	Operation Code	First Source Register	Second Source Register	Result Register	Shift Amount	Function Code

- Function code specifies which operation

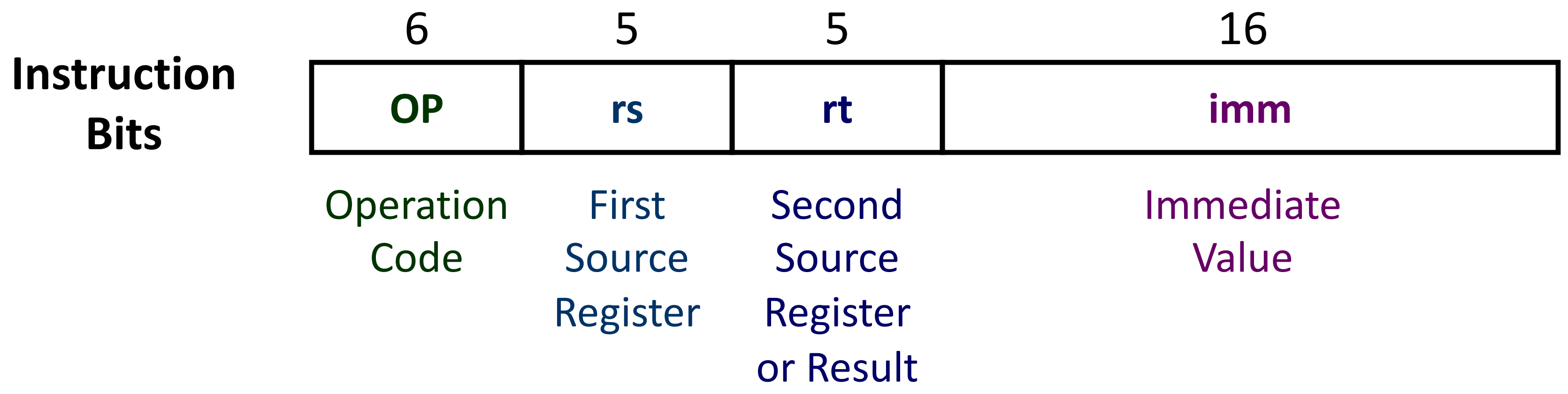
R – Format Limitations

- The R-Format works well for register ALU-type operations, but what about immediate or load-store instructions?

- Consider for example the lw instruction that takes an offset in addition to two registers
  - R-format would provide 5 bits for the offset
  - Offsets of only 32 are not all that useful!

I – Format

- The immediate instruction format
  - Uses different opcodes for each instruction
  - Immediate field is signed (positive/negative constants)
  - Used for loads and stores as well as other instructions with immediates (addi, lui, etc.)
  - Also used for branches



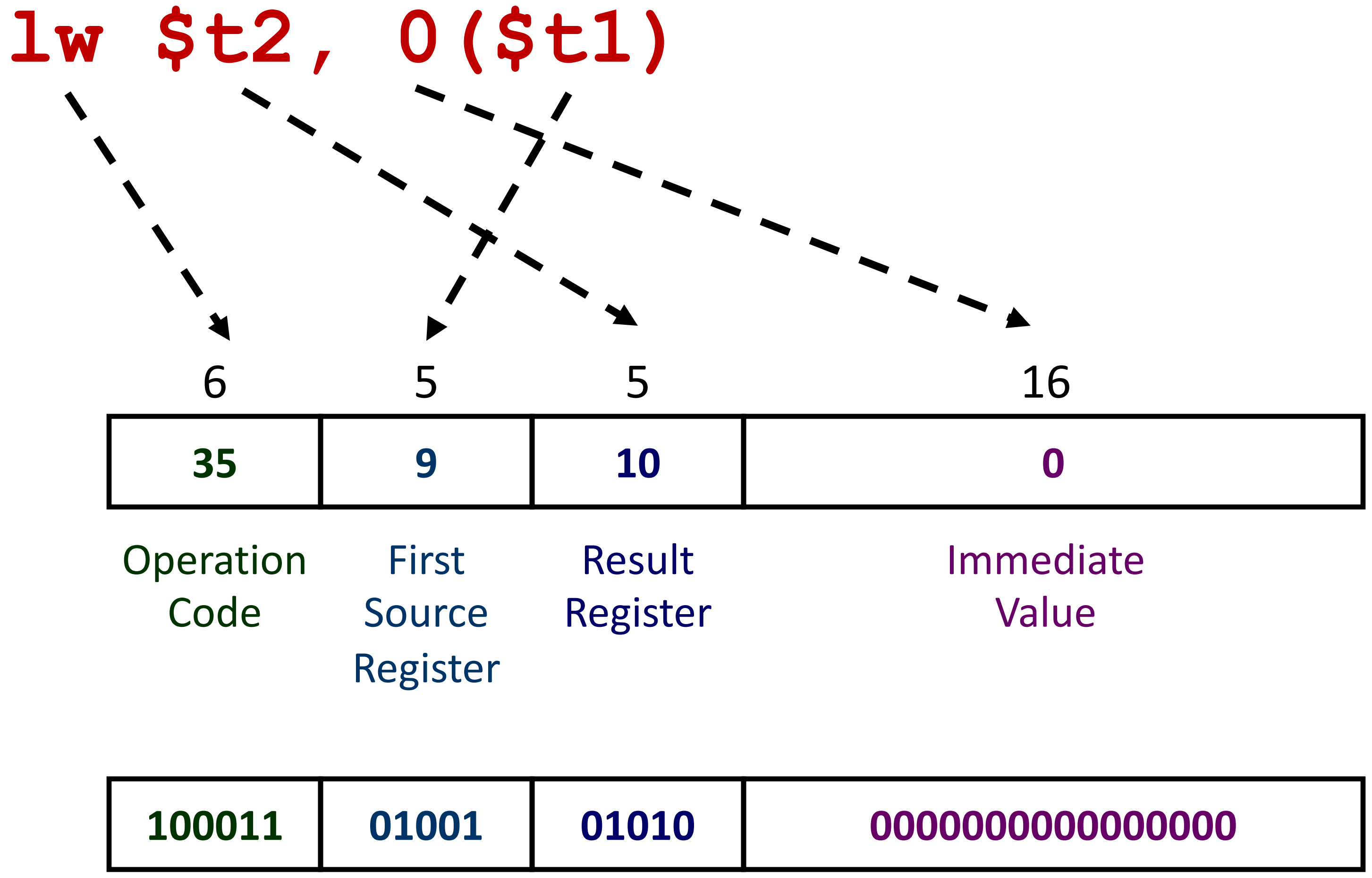
I – Format Example

- Consider the `lw` instruction

```
lw $t2, 0($t1)
```

I – Format Example

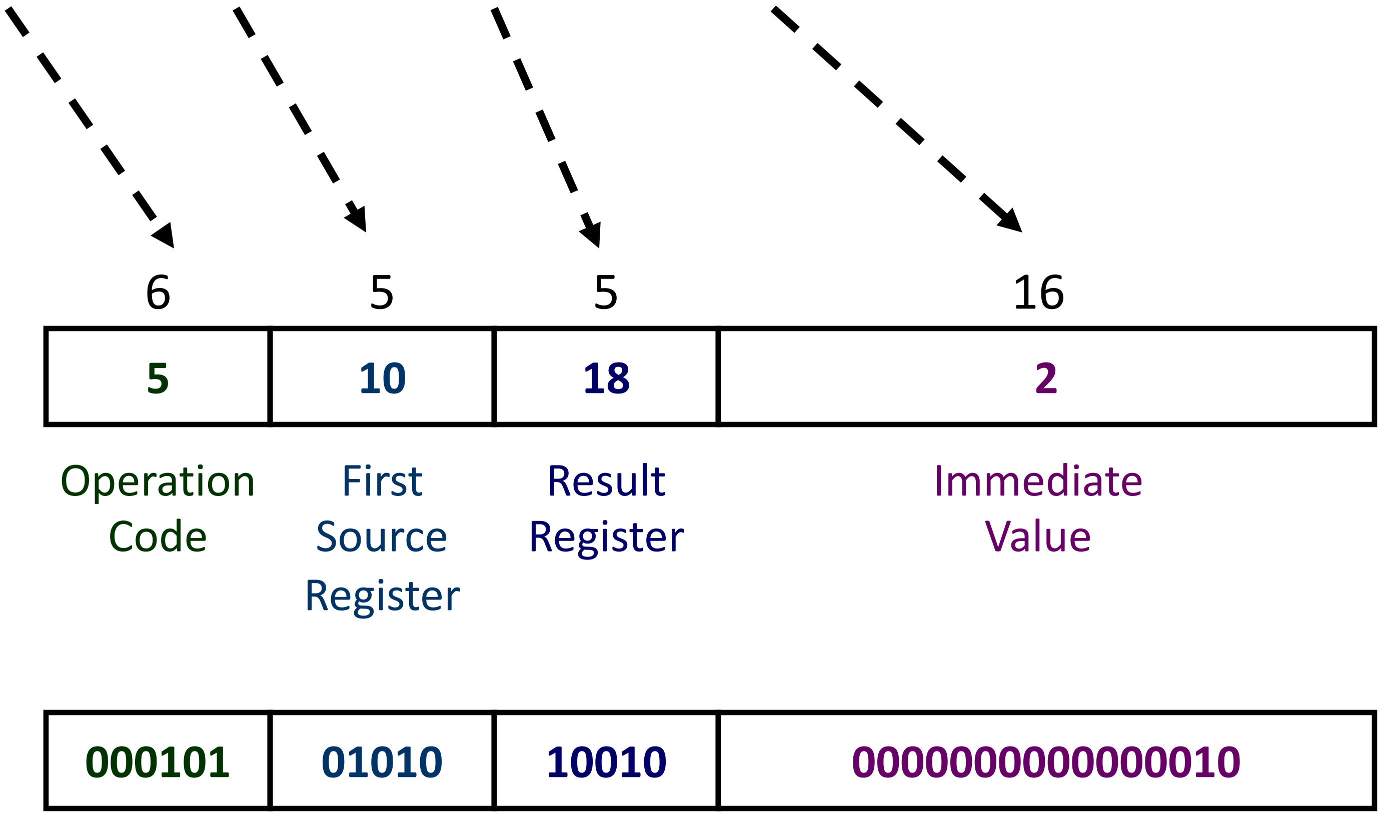
- Consider the `lw` instruction



I – Format Example

- Consider the `bne` instruction

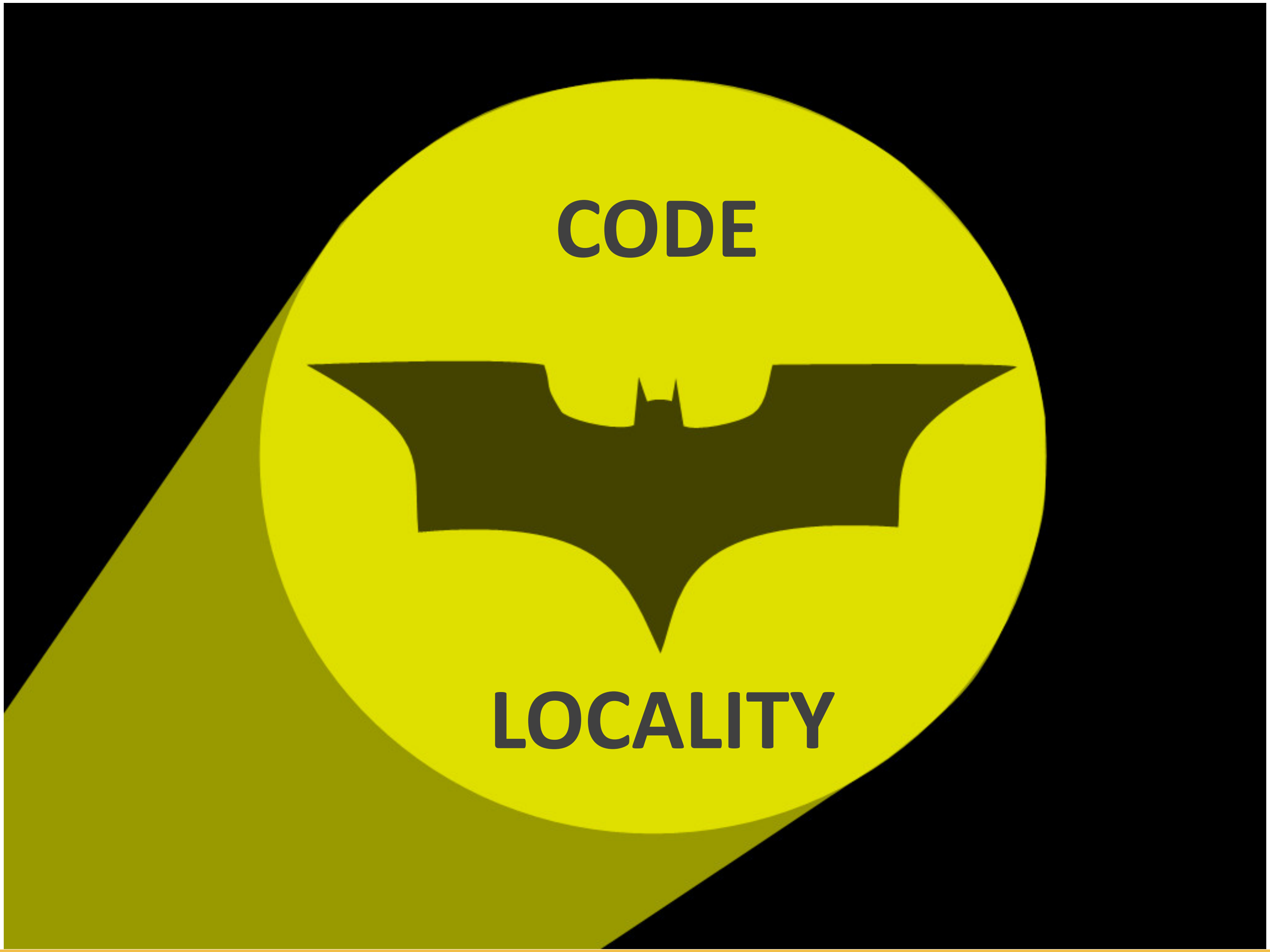
```
bne $t2, $s2, Exit
```



Branch Addresses

- How should we make use of immediate addresses when branching?
- What do we know?
  - 16-bit immediate can specify with block of 2^{16} (64k) instructions
  - 16 bits too small for many programs
    - All instructions are 32 bits or 4 bytes → all instruction addresses are multiples of 4 (two LSBs must be 0)
    - Address space can name $2^{32}/2^{2} = 2^{30}$ words
    - Branch immediates can only cover $2^{16}/2^{30} = 1/2^{14}$th of the address space!
- But what do we know about programs?

PC Relative Addressing



PC Relative Addressing

- Loops and other structured programming constructs tend to be relatively small
- Therefore, we actually branch to the address that is

$$\text{PC} + 4 + 4 \times \texttt{immediate}$$

I – Format Example (cont.)

- Consider the `bne` instruction

```
0x1000: bne   $t2, $s2, Exit
0x1004: addui $t3, $s3, 1
0x1008: lw    $s2, 0($t3)
0x100c: Exit: addui $s2, $s1, 10
0x1010: ...
```

Exit = 0x100c = PC + 4 + 4*immediate

immediate = (0x100c - PC - 4)/4 = 8/4 = 2

I – Format Example (cont.)

- Consider the **`bne`** instruction

```
0x1000:  bne    $t2, $s2, Exit
```

In-class Assessment!

Access Code: $uMMiT

Note: sharing access code to those outside of classroom or using access while outside of classroom is considered cheating

Exit = 0x100c = PC + 4 + 4*immediate

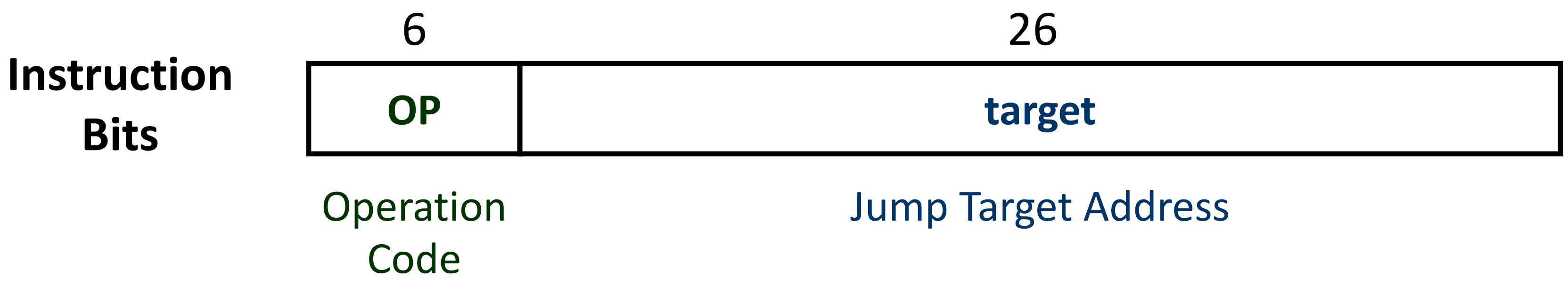
immediate = (0x100c - PC - 4)/4 = 8/4 = 2

I – Format Example (cont.)

- Verify assembling in MARS
  - Settings → “Delayed Branching” NOT selected

J – Format

- The jump instruction format
  - Different opcodes for each instruction (e.g. `j`, `jal`)
  - Absolute addressing since long jumps are common
  - Based on word addressing (target × 4)
  - Pseudodirect addressing, where 26 bits are from target and remaining 4 bits come from the upper bits of PC



$$\texttt{Jump PC} = \texttt{PC}_{31..28} \,||\, \texttt{target} \,||\, \texttt{00}$$

Branching Far, Far Away

- If the target is greater than -2^{15} to $2^{15}-1$ words away, then the compiler inverts the condition and inserts an unconditional jump
- Consider the example where L1 is far away

```
    beq $s0, $s1, L1    # goto L1 if $s0=$s1
```

- Can be rewritten as

```
    bne $s0, $s1, L2    # Inverted
    j L1             # Unconditional jump
L2:
```

- Compiler must be careful not to cross 256 MB boundaries with jump instructions

Instruction Format Summary

R – Format Instructions

6	5	5	5	5	6
OP=0	**rs**	**rt**	**rd**	**shamt**	**funct**

- Register ALU operations

I – Format Instructions

6	5	5	16
OP	**rs**	**rt**	**imm**

- Load/store, ALU immediate, conditional branch

J – Format Instructions

6	26
OP	**target**

- Jump, Jump-and-Link

sum_pow2 Revised Assembly

```
sum_pow2:                        # $a0 = b, $a1 = c
    addu $a0,$a0,$a1             # a = b + c, $a0 = a
    bltz $a0, Exceed             # goto Exceed if $a0 < 0
    slti $v0,$a0,8               # $v0 = a < 8
    beq $v0,$zero, Exceed        # goto Exceed if $v0 == 0
    addiu $v1,$sp,8              # $v1 = pow2 address
    sll $v0,$a0,2                # $v0 = a*4
    addu $v0,$v0,$v1             # $v0 = pow2 + a*4
    lw $v0,0($v0)                # $v0 = pow2[a]
    j Return                     # goto Return
Exceed:
    addu $v0,$zero,$zero         # $v0 = 0
Return:
    jr $ra                       # return sum_pow2
```

sum_pow2 Machine and Disassembly

```
sum_pow2:
0x400a98:  00 85 20 21   addu a0,a0,a1
0x400a9c:  04 80 00 07   bltz a0,0x400abc
0x400aa0:  28 82 00 06   slti v0,a0,6
0x400aa4:  10 40 00 05   beq v0,zero,0x400abc
0x400aa8:  27 a3 00 08   addiu v1,sp,8
0x400aac:  00 04 10 80   sll v0,a0,2
0x400ab0:  00 43 10 21   addu v0,v0,v1
0x400ab4:  8c 42 00 00   lw v0,0(v0)
0x400ab8:  10 00 00 01   j 0x400ac0
0x400abc:  00 00 10 21   addu v0,zero,zero
0x400ac0:  03 e0 00 08   jr ra
```

Addressing Modes Summary

- Register addressing
  - Operand is a register (e.g. ALU)
- Base/displacement addressing (ex. load/store)
  - Operand is at the memory location that is the sum of a base register + a constant
- Immediate addressing (e.g. constants)
  - Operand is a constant within the instruction itself
- PC-relative addressing (e.g. branch)
  - Address is the sum of PC and constant in instruction (e.g. branch)
- Pseudo-direct addressing (e.g. jump)
  - Target address is concatenation of field in instruction and the PC

Acknowledgments

- These slides contain material developed and copyright by:
  - Joe Zambreno (Iowa State)
  - David Patterson (UC Berkeley)
  - Mary Jane Irwin (Penn State)
  - Christos Kozyrakis (Stanford)
  - Onur Mutlu (Carnegie Mellon)
  - Krste Asanović (UC Berkeley)