CprE 381: Computer Organization and Assembly Level Programming

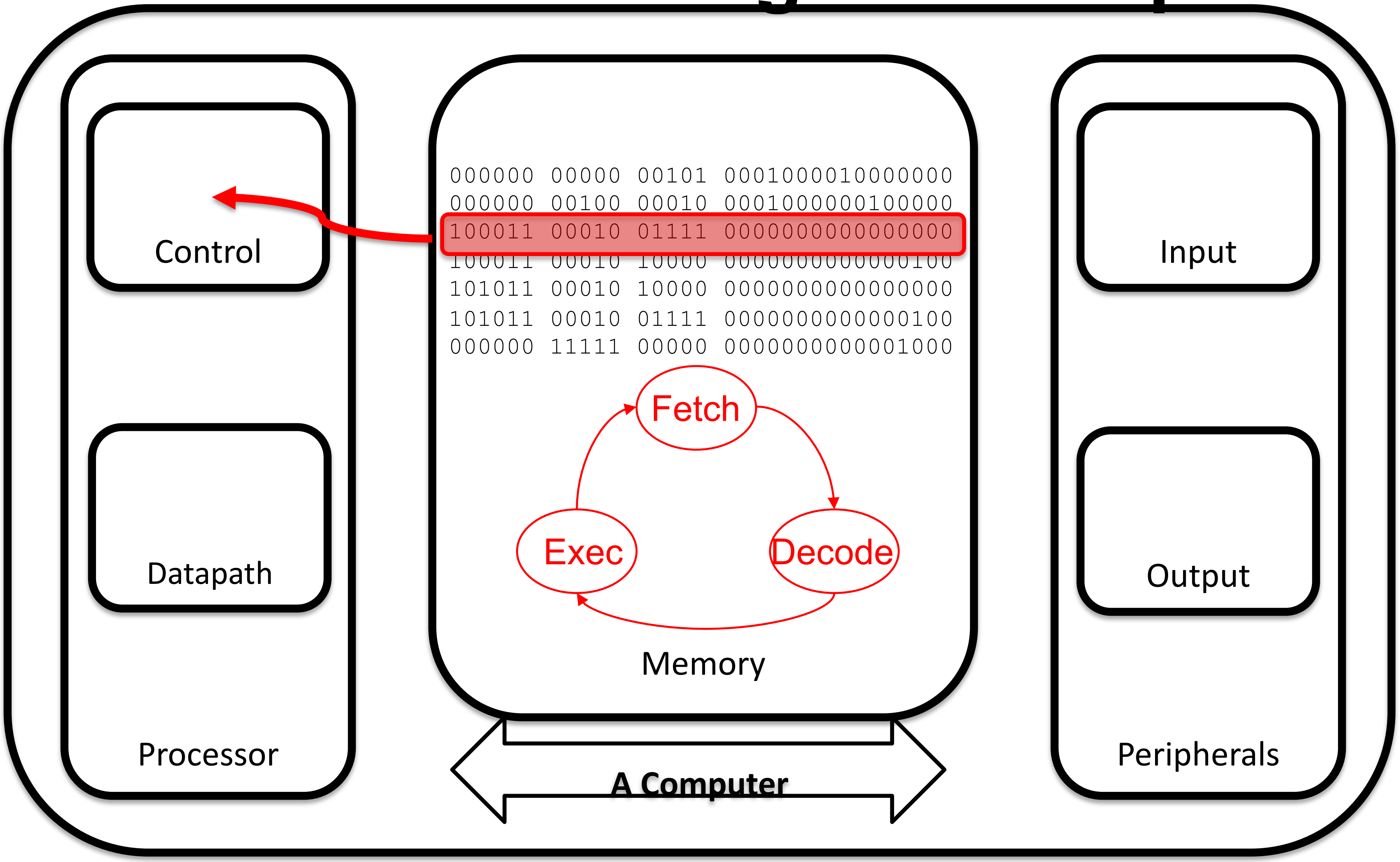
MIPS Machine Code

Henry Duwe
Electrical and Computer Engineering
Iowa State University

Administrative

- Labs 1-4
  - Learn a little VHDL to use in your term project
  - Review Digital Logic components + implement in VHDL for use in term project
- Lab 3 due BY START OF LAB SECTION
  - There will be no mercy on late labs starting this week
- Select partners for term project
  - Canvas → People → Groups
- VDI
  - 89% of class can use
  - Encourage remaining 11% to seek help from ETG
- Exam 1 in 2 weeks – in class, multiple rooms

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 Simple Arithmetic

Instruction	Example	Meaning	Comments
add	`add $1,$2,$3`	`$1 = $2 + $3`	3 operands; Overflow
subtract	`sub $1,$2,$3`	`$1 = $2 – $3`	3 operands; Overflow
add immediate	`addi $1,$2,100`	`$1 = $2 + 100`	+ constant; Overflow
add unsigned	`addu $1,$2,$3`	`$1 = $2 + $3`	3 operands; No overflow
sub unsigned	`subu $1,$2,$3`	`$1 = $2 – $3`	3 operands; No overflow
add imm unsign	`addiu $1,$2,100`	`$1 = $2 + 100`	+ constant; No overflow

- Your task: check out logical and shift instructions

Review: MIPS Integer Load/Store

Instruction	Example	Meaning	Comments
store word	`sw $1,8($2)`	`Mem[8+$2]=$1`	Store word
store half	`sh $1,6($2)`	`Mem[6+$2]=$1`	Stores only lower 16b
store byte	`sb $1,5($2)`	`Mem[5+$2]=$1`	Stores only lowest byte
load word	`lw $1,8($2)`	`$1=Mem[8+$2]`	Load word
load halfword	`lh $1,6($2)`	`$1=Mem[6+$2]`	Load half; sign extend
load half unsign	`lhu $1,6($2)`	`$1=Mem[6+$2]`	Load half; zero extend
load byte	`lb $1,5($2)`	`$1=Mem[5+$2]`	Load byte; sign extend
load byte unsign	`lbu $1,5($2)`	`$1=Mem[5+$2]`	Load byte; zero extend

Review: MIPS Control Flow

Instruction	Example	Meaning
jump	`j L`	`goto L`
jump register	`jr $1`	`goto value in $1`
jump and link	`jal L`	`goto L and set $ra`
jump and link register	`jalr $1`	`goto $1 and set $ra`
branch equal	`beq $1,$2,L`	`if ($1 == $2) goto L`
branch not equal	`bne $1,$2,L`	`if ($1 != $2) goto L`
branch less than 0	`bltz $1,L`	`if ($1 < 0) goto L`
branch less than / eq 0	`blez $1,L`	`if ($1 <= 0) goto L`
branch greater than 0	`bgtz $1,L`	`if ($1 > 0) goto L`
branch greater than / eq 0	`bgez $1,L`	`if ($1 >= 0) goto L`

Review: MIPS Comparisons

Instruction	Example	Meaning	Comments
set less than	`slt $1,$2,$3`	`$1=($2<$3)`	Comp less than signed
set less than imm	`slti $1,$2,100`	`$1=($2<100)`	Comp w/const signed
set less unsgn	`sltu $1,$2,$3`	`$1=($2<$3)`	Comp less than unsigned
slt imm unsgn	`sltiu $1,$2,100`	`$1=($2<100)`	Comp w/const unsigned

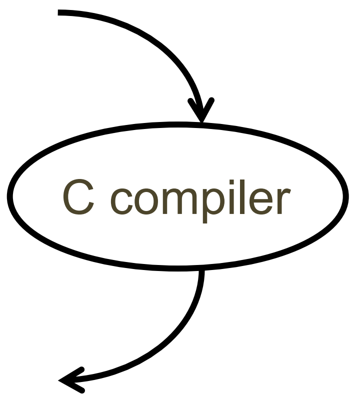
- C

```
 if (8 < a)goto Exceed
slti $v0, $a0, 9         # $v0 = $a0<9
beq $v0, $zero, Exceed # goto if $v0==0
```

MIPS Machine Code

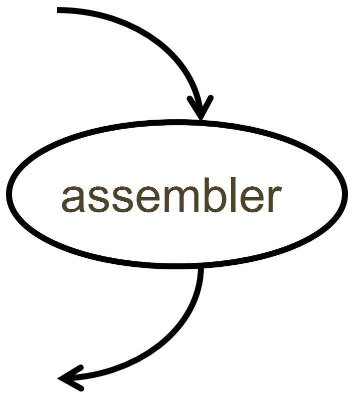
- High-level language program (in C)

```
swap (int v[], int k)
. . .
```



- Assembly language program (for MIPS)

```
swap:   sll    $2, $5, 2
        add    $2, $4, $2
        lw     $15, 0($2)
        lw     $16, 4($2)
        sw     $16, 0($2)
        sw     $15, 4($2)
        jr     $31
```



- Machine (object) code (for MIPS)

```
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
```

Machine Language Representation

- Instructions are represented as binary data in memory
    - "Stored program" - Von Neumann
  - Simplicity
    - One memory system
    - Same addresses used for branches, procedures, data, etc.
  - The only difference is how bits are interpreted
    - What are the risks of this decision?

- Binary (backwards) compatibility
  - Commercial software relies on ability to work on next generation hardware (E.g., take advantage of Moore's Law)
  - This leads to a (possibly) very long life for an ISA
    - x86 turned 40 last year

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

R – Format

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

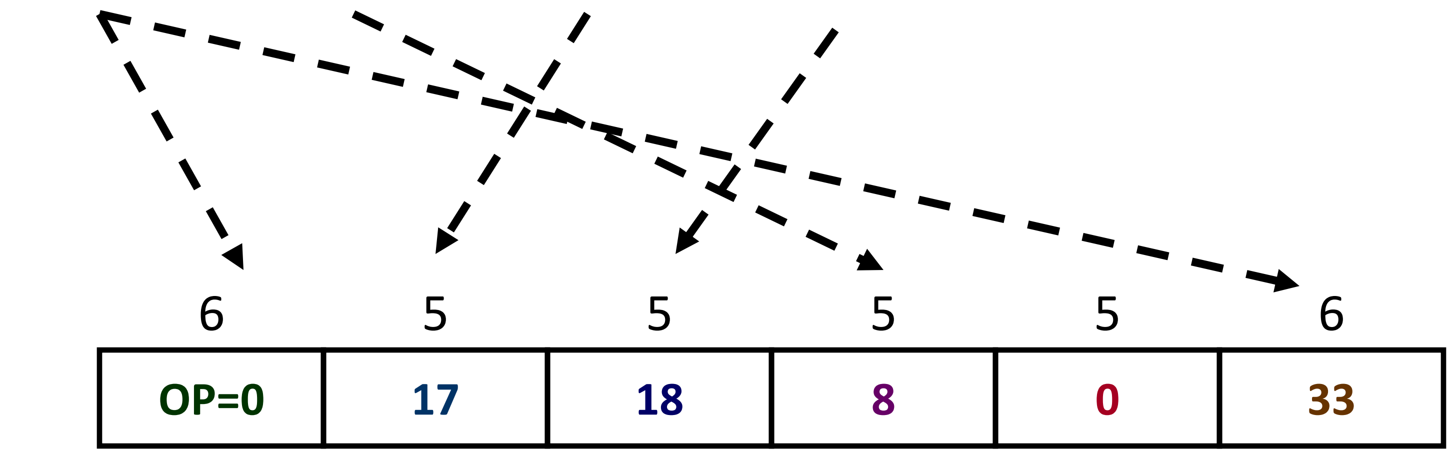
	6	5	5	5	5	6
Instruction Bits	**OP=0**	**rs**	**rt**	**rd**	**shamt**	**funct**
	Operation Code	First Source Register	Second Source Register	Result Register	Shift Amount	Function Code

- Function code specifies which operation

R – Format Example

- Consider the `addu` instruction

```
addu $t0, $s1, $s2
```



6	5	5	5	5	6
OP=0	**17**	**18**	**8**	**0**	**33**
Operation Code	First Source Register	Second Source Register	Result Register	Shift Amount	Function Code

000000	**10001**	**10010**	**01000**	**00000**	**100001**

R – Format Example

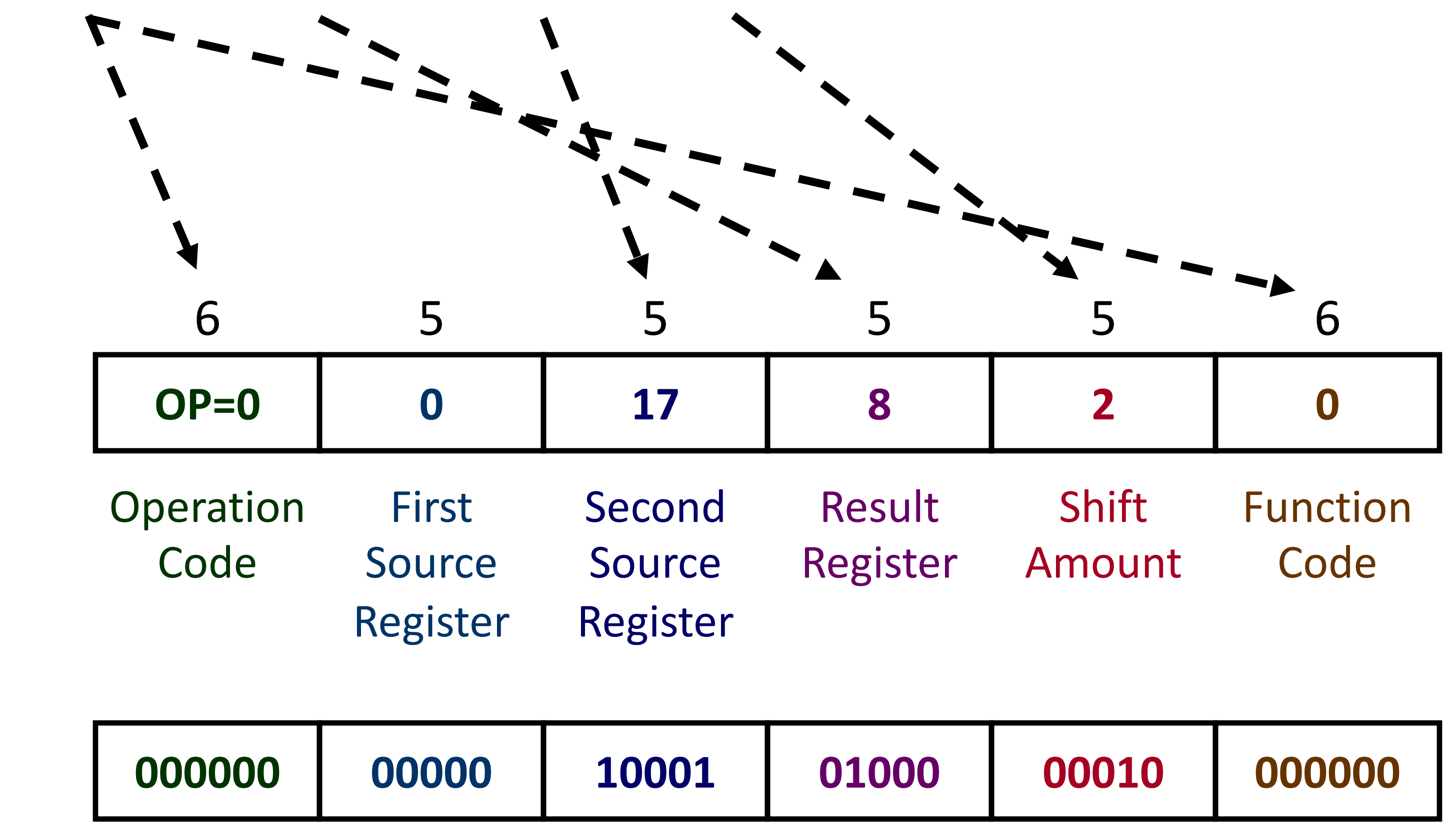
- Consider the `sll` instruction

```
sll $t0, $s1, 2
```

R – Format Example

- Consider the `sll` instruction

```
sll $t0, $s1, 2
```



6	5	5	5	5	6
OP=0	**0**	**17**	**8**	**2**	**0**
Operation Code	First Source Register	Second Source Register	Result Register	Shift Amount	Function Code
000000	**00000**	**10001**	**01000**	**00010**	**000000**

R – Format Example

- Consider the `sll` instruction

```
sll $t0, $s1, 2
```

In-class Assessment!

Access Code: iRobot

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

Operation Code	First Source Register	Second Source Register	Result Register	Shift Amount	Function Code
000000	**00000**	**10001**	**01000**	**00010**	**000000**

R – Format Example

- Decode the following instruction

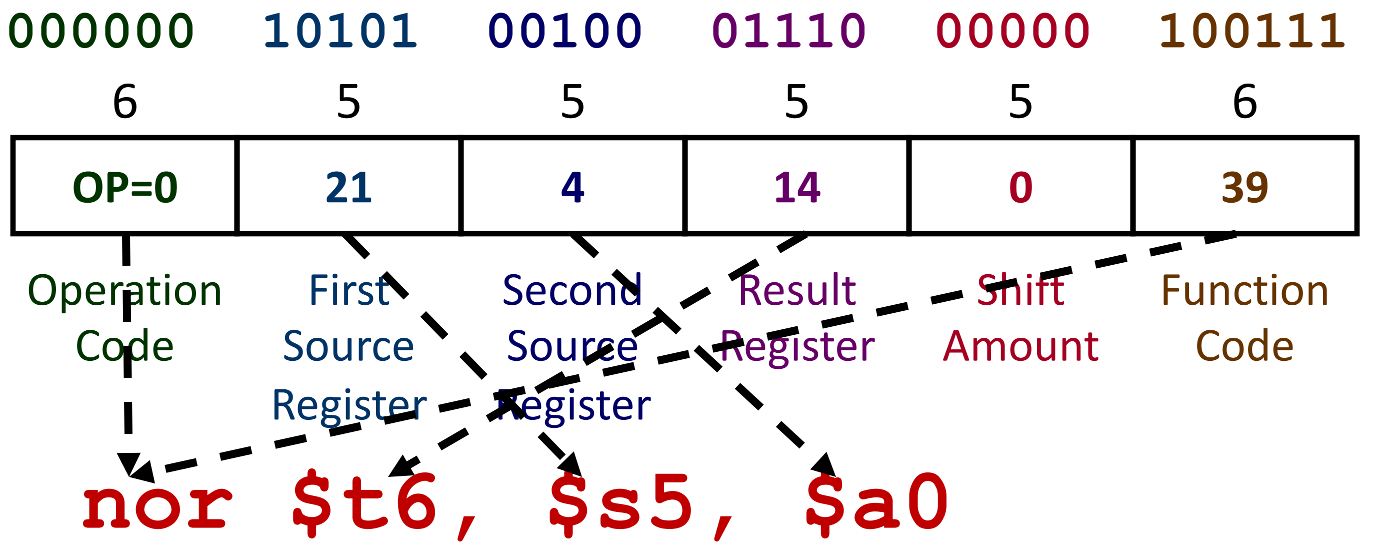
  0x**02a47027**

R – Format Example

- Decode the following instruction

  0x02a47027

```
0000 0010 1010 0100 0111 0000 0010 0111
```



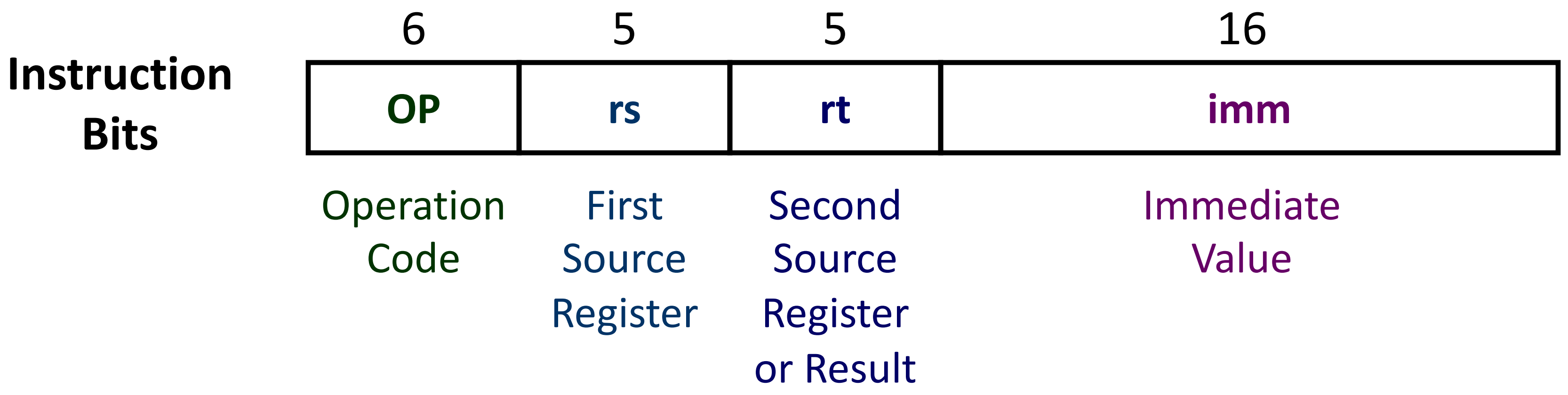
R – Format Limitations

- 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!

Preview: 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



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)