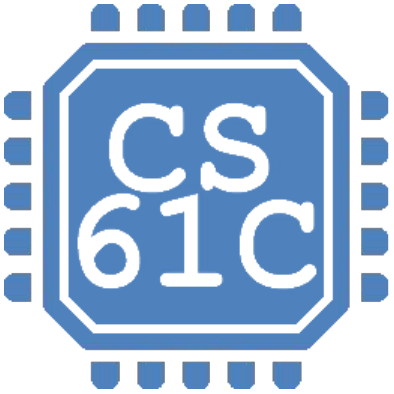


Great Ideas in Computer Architecture

MIPS Instruction Formats

Instructor: Justin Hsia

RISC-V Offers Simple, Modular ISA

"RISC-V is a new general-purpose instruction-set architecture (ISA) that's BSD licensed, extensible, and royalty free. It's clean and modular with a 32-, 64-, or 128-bit integer base and various optional extensions (e.g., floating point). RISC-V is easier to implement than some alternatives—minimal RISC-V cores are roughly half the size of equivalent ARM cores—and the ISA has already gathered some support from the semiconductor industry.

"Krste Asanovic, Andrew Waterman, and Yunsup Lee developed the RISC-V architecture at UC Berkeley, with input from David Patterson."

- Online article

Review of Last Lecture

- New registers: `$a0-$a3`, `$v0-$v1`, `$ra`, `$sp`
  - Also: `$at`, `$k0-k1`, `$gp`, `$fp`, `PC`
- New instructions: `slt`, `la`, `li`, `jal`, `jr`
- Saved registers: `$s0-$s7`, `$sp`, `$ra`
  Volatile registers: `$t0-$t9`, `$v0-$v1`, `$a0-$a3`
  - CalleR saves volatile registers it is using before making a procedure call
  - CalleE saves saved registers it intends to use

Question: Which statement below is TRUE about converting the following C code to MIPS?

```
int factorial(int n) {
  if(n == 0) return 1;
  else return(n*factorial(n-1));
}
```

(A) This function must be implemented recursively

(B) We can write this function without using any saved or temporary registers

(C) We must save `$ra` on the stack since we need to know where to return to

(D) We could copy `$a0` to `$a1` to store `n` across recursive calls instead of saving it

Great Idea #1: Levels of Representation/Interpretation

Higher-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

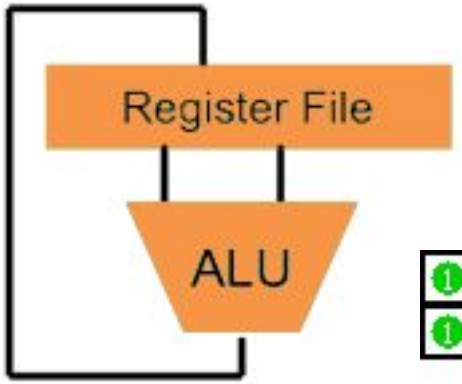
Logic Circuit Description (Circuit Schematic Diagrams)

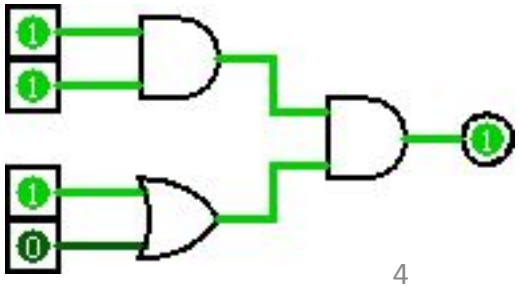
```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

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

We are here

```
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
```





Agenda

- Stored-Program Concept
- R-Format
- Administrivia
- I-Format
  - Branching and PC-Relative Addressing
- J-Format
- Bonus: Assembly Practice
- Bonus: Disassembly Practice

Big Idea: Stored-Program Concept

- Encode your instructions as binary data
  - 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
- Stored in memory, so both instructions and data words have addresses
  - Use with jumps, branches, and loads

Binary Compatibility

- Programs are distributed in binary form
  - Programs bound to specific instruction set
  - i.e. different versions for (old) Macs vs. PCs
- New machines want to run old programs ("binaries") as well as programs compiled to new instructions
- Leads to "backward compatible" instruction sets that evolve over time
  - The selection of Intel 80x86 in 1981 for 1st IBM PC is major reason latest PCs still use 80x86 instruction set (Pentium 4); you 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 in length
  - `lw` and `sw` both access one word of memory
- 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, let instructions be in words, too

Instructions as Numbers (2/2)

- Divide the 32 bits of an instruction into "fields"
  - Each field tells the processor something about the instruction
  - Could use different fields for every instruction, but regularity leads to simplicity
- Define 3 types of *instruction formats:*
  - R-Format
  - I-Format
  - J-Format

Instruction Formats

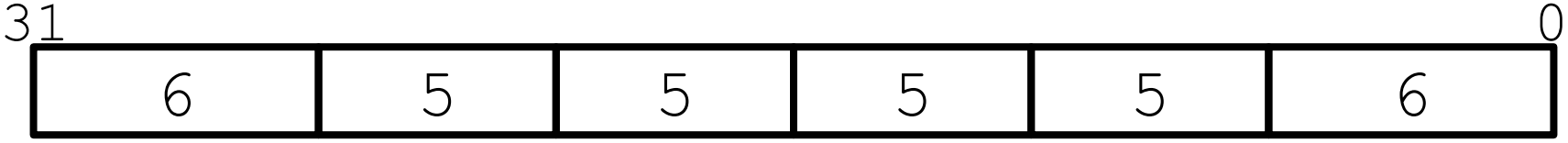
- I-Format: instructions with immediates, `lw`/`sw` (offset is immediate), and `beq`/`bne`
  - But not the shift instructions
- J-Format: `j` and `jal`
  - But not `jr`
- R-Format: all other instructions

Agenda

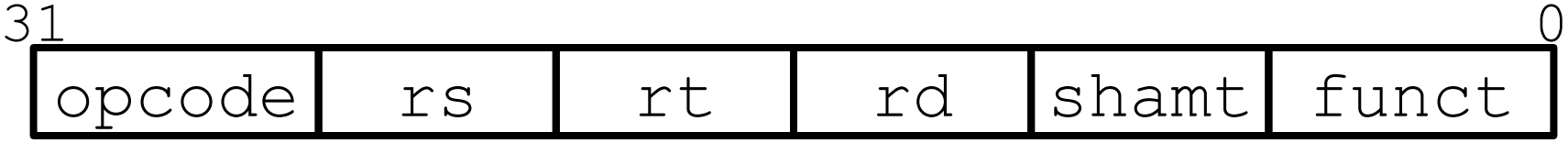
- Stored-Program Concept
- R-Format
- Administrivia
- I-Format
  - Branching and PC-Relative Addressing
- J-Format
- Bonus: Assembly Practice
- Bonus: Disassembly Practice

R-Format Instructions (1/4)

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



- Each field has a name:



- Each field is viewed as its own unsigned int
  - 5-bit fields can represent any number 0-31, while 6-bit fields can represent any number 0-63

R-Format Instructions (2/4)

- `opcode` (6): partially specifies operation
  - Set at 0b000000 for all R-Format instructions
- `funct` (6): combined with `opcode`, this number exactly specifies the instruction
- How many R-format instructions can we encode?
  - `opcode` is fixed, so 64
- Why aren't these a single 12-bit field?
  - We'll answer this later

R-Format Instructions (3/4)

- `rs` (5): specifies register containing 1st operand ("source register")
- `rt` (5): specifies register containing 2nd operand ("target register" – name is misleading)
- `rd` (5): specifies register that receives the result of the computation ("destination register")
- **Recall:** MIPS has 32 registers
  - Fit perfectly in a 5-bit field (use register numbers)
- These map intuitively to instructions
  - e.g. `add dst,src1,src2` → `add rd,rs,rt`
  - Depending on instruction, field may not be used

R-Format Instructions (4/4)

- `shamt` (5): The amount a shift instruction will shift by
  - Shifting a 32-bit word by more than 31 is useless
  - This field is set to `0` in all but the shift instructions
- For a detailed description of field usage and instruction type for each instruction, see the MIPS Green Card

R-Format Example (1/2)

- MIPS Instruction:

```
add   $8,$9,$10
```

- Pseudo-code ("OPERATION" column):

```
add   R[rd] = R[rs] + R[rt]
```

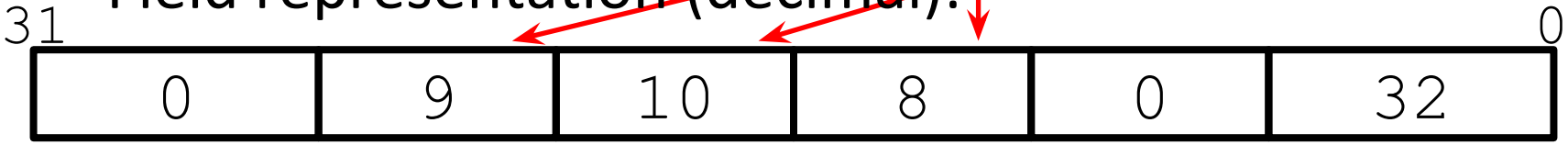
- Fields:

`opcode` = 0 (look up on Green Sheet)
`funct` = 32 (look up on Green Sheet)
`rd` = 8 (destination)
`rs` = 9 (first *operand*)
`rt` = 10 (second *operand*)
`shamt` = 0 (not a shift)

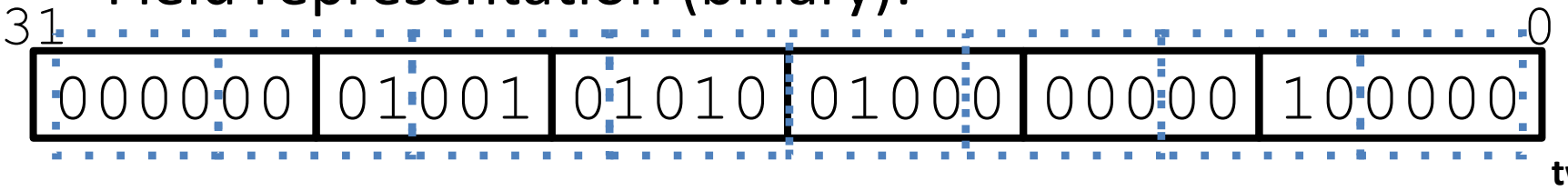
R-Format Example (2/2)

- MIPS Instruction: `add $8,$9,$10`

Field representation (decimal):



Field representation (binary):



hex representation: `0x012A4020`

decimal representation: `19,546,144`

Called a Machine Language Instruction

NOP (or NOOP)

- What is the instruction `0x00000000`?
  - `opcode` is 0, so is an R-Format
- Using Green Sheet, translates into:

```
sll $0,$0,0
```

  - What does this do? Nothing!
- This is a special instruction called `nop` (or `noop`) for "No Operation Performed"
  - We'll see its uses later in the course

Agenda

- Stored-Program Concept
- R-Format
- Administrivia
- I-Format
  - Branching and PC-Relative Addressing
- J-Format
- Bonus: Converting to Machine Code Practice

Administrivia

- Long weekend due to July 4, but HW1 and Proj1 due Sunday, July 3
- HW2 released tonight
- Project 2 released Thu/Fri, due 7/10
- Midterm 1 is Thu 7/7 in class

- Can play assembly game Human Resource Machine on lab machines by running `~cs61c/bin/hrm`

Agenda

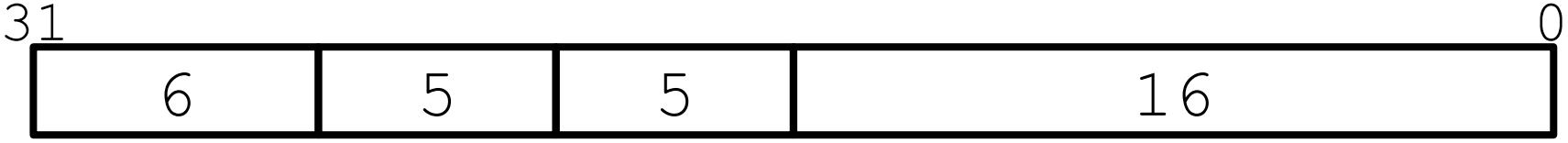
- Stored-Program Concept
- R-Format
- Administrivia
- I-Format
  - Branching and PC-Relative Addressing
- J-Format
- Bonus: Assembly Practice
- Bonus: Disassembly Practice

I-Format Instructions (1/4)

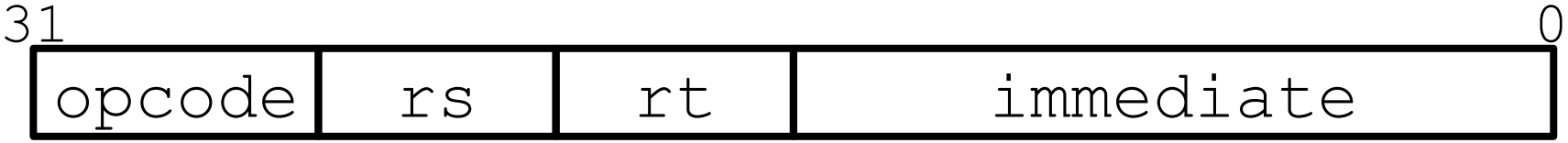
- What about instructions with immediates?
  - 5- and 6-bit fields too small for most immediates
- Ideally, MIPS would have only one instruction format (for simplicity)
  - Unfortunately here we need to compromise
- Define new instruction format that is *partially* consistent with R-Format
  - First 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



- Field names:



- **Key Concept:** Three fields are consistent with R-Format instructions
  - Most importantly, `opcode` is still in same location

I-Format Instructions (3/4)

- `opcode` (6): uniquely specifies the instruction
  - All I-Format instructions have non-zero `opcode`
  - R-Format has two 6-bit fields to identify instruction for consistency across formats
- `rs` (5): specifies a register operand
  - Not always used
- `rt` (5): specifies register that receives result of computation ("target register")
  - Name makes more sense for I-Format instructions

I-Format Instructions (4/4)

- `immediate` (16): *two's complement* number
  - All computations done in words, so 16-bit immediate must be *extended* to 32 bits
  - Green Sheet specifies ZeroExtImm or SignExtImm based on instruction
- Can represent 2^{16} different immediates
  - This is large enough to handle the offset in a typical `lw`/`sw`, plus the vast majority of values for `slti`

I-Format Example (1/2)

- MIPS Instruction:

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

- Pseudo-code ("OPERATION" column)

```
addi  R[rt] = R[rs] + SignExtImm
```

- Fields:

`opcode` = 8 (look up on Green Sheet)

`rs` = 22 (register containing operand)

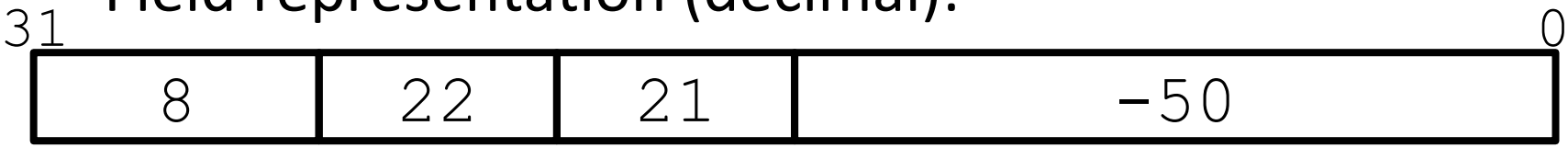
`rt` = 21 (target register)

`immediate` = -50 (decimal by default, can also be specified in hex)

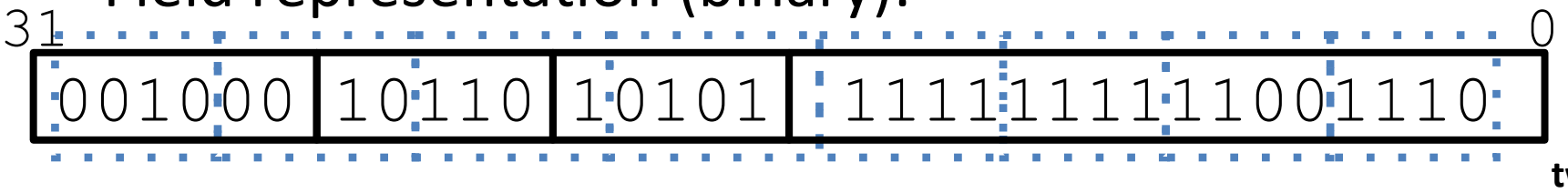
I-Format Example (2/2)

- MIPS Instruction: `addi $21,$22,-50`

Field representation (decimal):



Field representation (binary):



hex representation: `0x22D5FFCE`

decimal representation: `584,449,998`

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

OPCODE/FUNCT:

```
subu   0/35
lw    35/--
addi   8/--
```

Register names and numbers:

```
   0: $0
 8-15: $t0-$t7
16-23: $s0-$s7
```

(A) `subu $s0,$s0,$s0`

(B) `lw $0,0($0)`

(C) `addi $0,$0,35`

(D) `subu $0,$0,$0`

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 31-16) of `reg` and zeros the lower half (bits 15-0)

`lui` Example

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

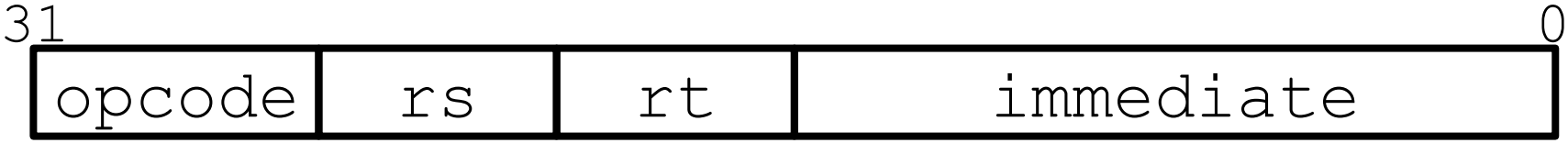
```
   lui $at,0xABAB      # upper 16
ori $at,$at,0xCDCD # lower 16
add $t0,$t0,$at     # move
```

Only the assembler gets to use $at

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

Branching Instructions

- `beq` and `bne`
  - Need to specify an address to go to
  - Also take 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

- So just how much of memory can we reach?

Branching Reach

- **Recall:** MIPS uses 32-bit addresses
  - Memory is byte-addressed
- Instructions are *word-aligned*
  - Address is always a multiple of 4 (in bytes), meaning it ends with `0b00` in binary
  - Number of bytes to add to the PC will always be a multiple of 4
- Immediate specifies words instead of bytes
  - Can now branch ± 2^{15} words
  - We can reach 2^{16} instructions = 2^{18} bytes around 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

Branch Example (1/2)

- MIPS Code:

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

Start counting from instruction AFTER the branch

- I-Format fields:

  opcode = 4 (look up on Green Sheet)

  rs = 9 (first operand)

  rt = 0 (second operand)

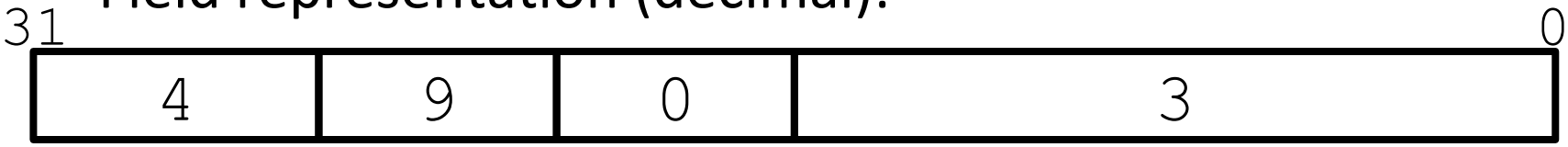
  immediate: 3 ?

Branch Example (2/2)

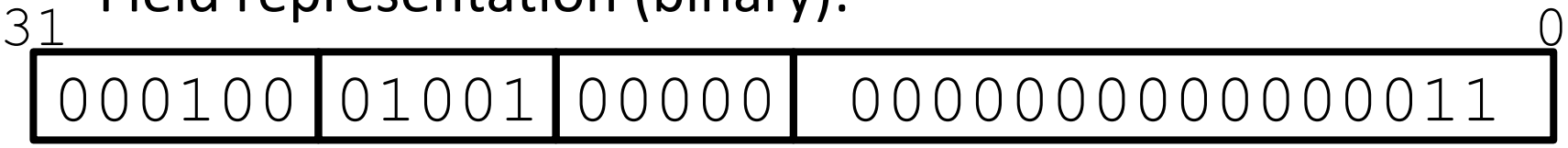
- MIPS Code:

```
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
```

Get To Know Your Staff

- Category: **Games**

Agenda

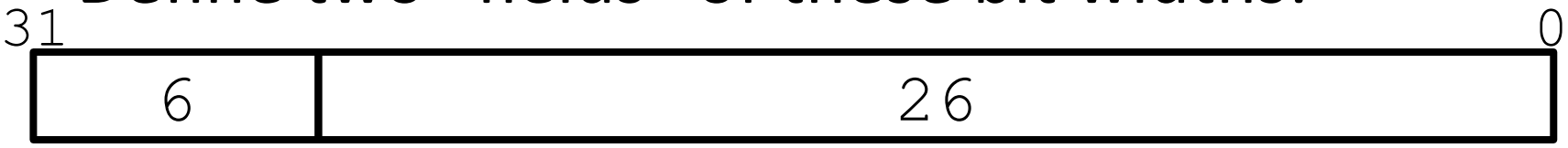
- Stored-Program Concept
- R-Format
- Administrivia
- I-Format
  - Branching and PC-Relative Addressing
- J-Format
- Bonus: Assembly Practice
- Bonus: Disassembly Practice

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 256MiB 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, 0b00 }
- 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

Question: When combining two C files into one executable, we can compile them independently and then merge them together.

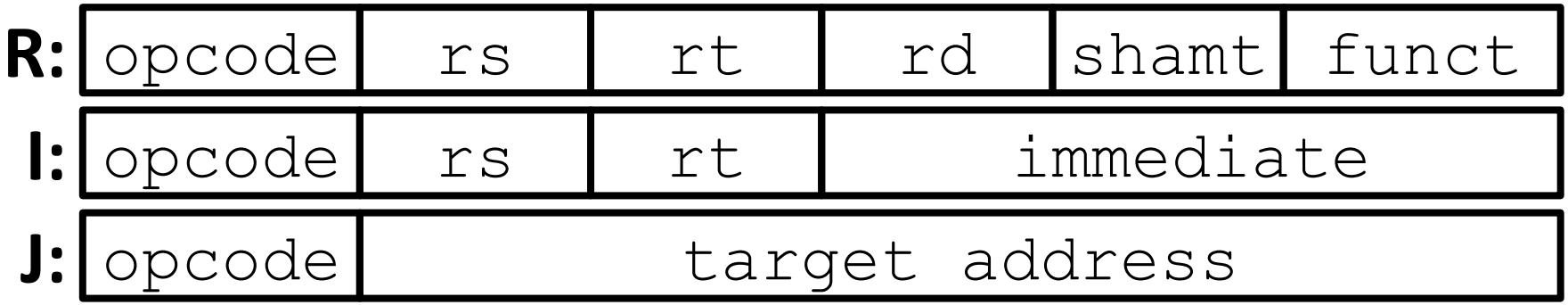
When merging two or more binaries:

1) Jump instructions don't require any changes
2) Branch instructions don't require any changes

	1	2
(A)	F	F
(B)	F	T
(C)	T	F
(D)	T	T

Summary

- The Stored Program concept is very powerful
  - Instructions can be treated and manipulated the same way as data in both hardware and software
- MIPS Machine Language Instructions:



- Branches use PC-relative addressing, Jumps use absolute addressing

BONUS SLIDES

You are responsible for the material contained on the following slides, though we may not have enough time to get to them in lecture.

They have been prepared in a way that should be easily readable and the material will be touched upon in the following lecture.

Agenda

- Stored-Program Concept
- R-Format
- Administrivia
- I-Format
  - Branching and PC-Relative Addressing
- J-Format
- Bonus: Assembly Practice
- Bonus: Disassembly Practice

Assembly Practice

- Assembly is the process of converting assembly instructions into machine code
- On the following slides, there are 6-lines of assembly code, along with space for the machine code
- For each instruction,
  1) Identify the instruction type (R/I/J)
  2) Break the space into the proper fields
  3) Write field values in decimal
  4) Convert fields to binary
  5) Write out the machine code in hex
- Use your Green Sheet; answers follow

Code Questions

```
Addr   Instruction
 800   Loop: sll $t1,$s3,2

 804   addu  $t1,$t1,$s6

 808   lw    $t0,0($t1)

 812   beq   $t0,$s5, Exit

 816   addiu $s3,$s3,1

 820   j     Loop

       Exit:
```

Material from past lectures:
What type of C variable is probably stored in $s6?

Write an equivalent C loop using a→$s3, b→$s5, c→$s6. Define variable types (assume they are initialized somewhere) and feel free to introduce other variables as you like.

In English, what does this loop do?

Code Questions

```
Addr   Instruction
 800   Loop: sll $t1,$s3,2

 804   addu  $t1,$t1,$s6

 808   lw    $t0,0($t1)

 812   beq   $t0,$s5, Exit

 816   addiu $s3,$s3,1

 820   j     Loop

       Exit:
```

Material from past lectures:

What type of C variable is probably stored in $s6?

int * (or any pointer)

Write an equivalent C loop using a→$s3, b→$s5, c→$s6. Define variable types (assume they are initialized somewhere) and feel free to introduce other variables as you like.

```
int a,b,*c;
/* values initialized */
while(c[a] != b)  a++;
```

In English, what does this loop do?

Finds an entry in array c that matches b.

Assembly Practice Question

Addr	Instruction
800	Loop: sll $t1,$s3,2
__:	
804	addu $t1,$t1,$s6
__:	
808	lw $t0,0($t1)
__:	
812	beq $t0,$s5, Exit
__:	
816	addiu $s3,$s3,1
__:	
820	j Loop
__:	
	Exit:

Assembly Practice Answer (1/4)

Addr Instruction

```
 800  Loop: sll $t1,$s3,2
```

R:

opcode	rs	rt	rd	shamt	funct

```
 804  addu  $t1,$t1,$s6
```

R:

opcode	rs	rt	rd	shamt	funct

```
 808  lw    $t0,0($t1)
```

I:

opcode	rs	rt	immediate

```
 812  beq   $t0,$s5, Exit
```

I:

opcode	rs	rt	immediate

```
 816  addiu $s3,$s3,1
```

I:

opcode	rs	rt	immediate

```
 820  j     Loop
```

J:

opcode	target address

```
      Exit:
```

Assembly Practice Answer (2/4)

Addr Instruction

800 Loop: sll $t1,$s3,2

R: | 0 | 0 | 19 | 9 | 2 | 0 |

804 addu $t1,$t1,$s6

R: | 0 | 9 | 22 | 9 | 0 | 33 |

808 lw $t0,0($t1)

I: | 35 | 9 | 8 | 0 |

812 beq $t0,$s5, Exit

I: | 4 | 8 | 21 | 2 |

816 addiu $s3,$s3,1

I: | 8 | 19 | 19 | 1 |

820 j Loop

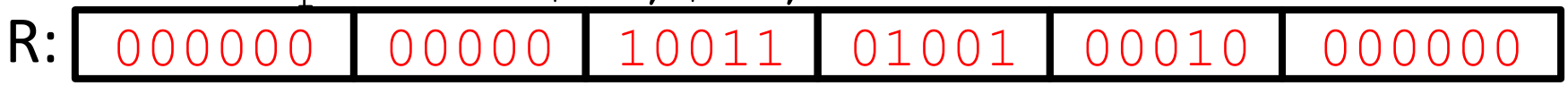
J: | 2 | 200 |

Exit:

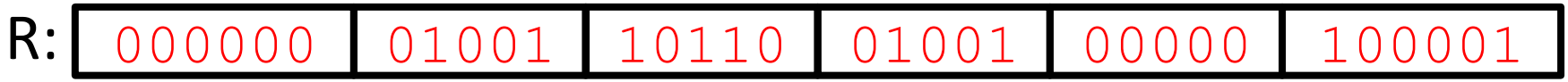
Assembly Practice Answer (3/4)

Addr Instruction

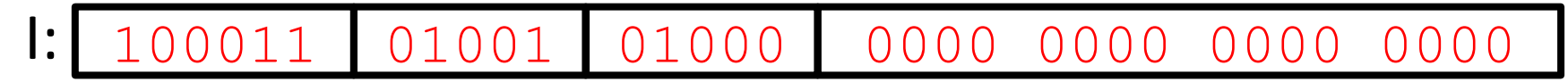
```
 800  Loop: sll  $t1,$s3,2
```



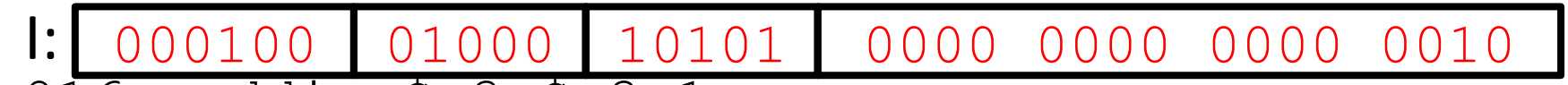

```
 804  addu  $t1,$t1,$s6
```



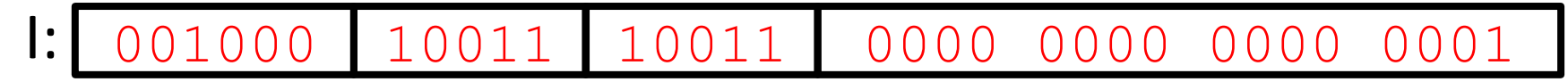

```
 808  lw    $t0,0($t1)
```



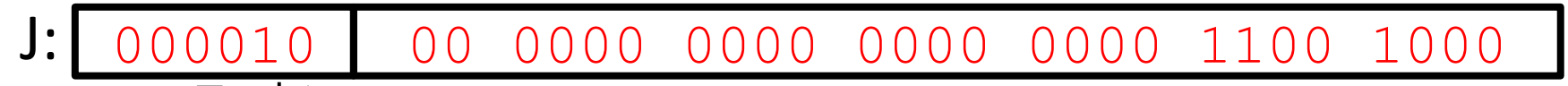

```
 812  beq   $t0,$s5, Exit
```




```
 816  addiu $s3,$s3,1
```




```
 820  j     Loop
```




```
      Exit:
```

Assembly Practice Answer (4/4)

```
Addr   Instruction
 800   Loop: sll $t1,$s3,2
 R:    0x 0013 4880
 804   addu  $t1,$t1,$s6
 R:    0x 0136 4821
 808   lw    $t0,0($t1)
 I:    0x 8D28 0000
 812   beq   $t0,$s5, Exit
 I:    0x 1115 0002
 816   addiu $s3,$s3,1
 I:    0x 2273 0001
 820   j     Loop
 J:    0x 0800 00C8
       Exit:
```

Agenda

- Stored-Program Concept
- R-Format
- Administrivia
- I-Format
  - Branching and PC-Relative Addressing
- J-Format
- Bonus: Assembly Practice
- Bonus: Disassembly Practice

Disassembly Practice

- Disassembly is the opposite process of figuring out the instructions from the machine code
- On the following slides, there are 6-lines of machine code (hex numbers)
- Your task:
  1) Convert to binary
  2) Use `opcode` to determine format and fields
  3) Write field values in decimal
  4) Convert fields MIPS instructions (try adding labels)
  5) Translate into C (be creative!)
- Use your Green Sheet; answers follow

Disassembly Practice Question

Address Instruction

```
0x00400000   0x00001025
...  0x0005402A
   0x11000003
   0x00441020
   0x20A5FFFF
   0x08100001
```

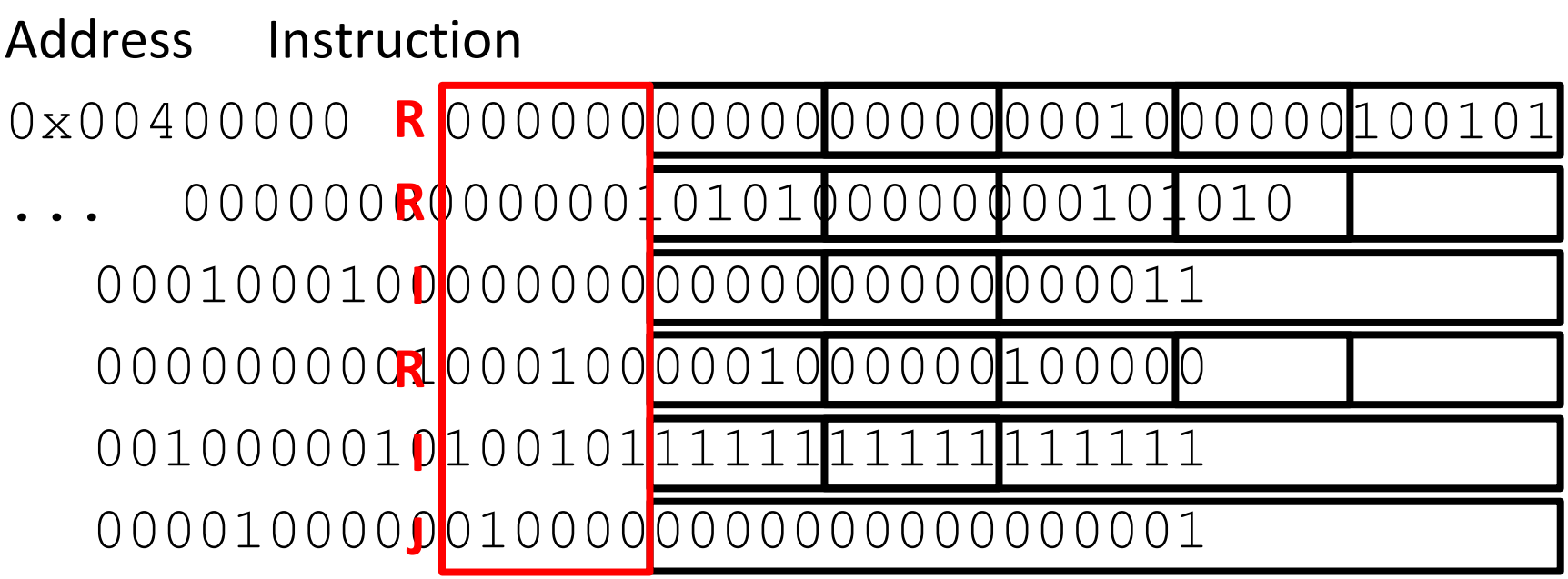
Disassembly Practice Answer (1/9)

Address Instruction

```
0x00400000   00000000000000000001000000100101
...   00000000000001010100000000101010
      00010001000000000000000000000011
      00000000010001000001000000100000
      00100000101001011111111111111111
      00001000000100000000000000000001
```

1) Converted to binary

Disassembly Practice Answer (2/9)



2) Check `opcode` for format and fields...

– 0 (R-Format), 2 or 3 (J-Format), otherwise (I-Format)

Disassembly Practice Answer (3/9)

Address	Instruction						
0x00400000	R	0	0	0	2	0	37
...	R	0	0	5	8	0	42
	I	4	8	0	+3		
	R	0	2	4	2	0	32
	I	8	5	5	-1		
	J	2	0x0100001				

3) Convert to decimal

– Can leave target address in hex

Disassembly Practice Answer (4/9)

```
Address     Instruction
0x00400000   or   $2,$0,$0
0x00400004   slt  $8,$0,$5
0x00400008   beq  $8,$0,3
0x0040000C   add  $2,$2,$4
0x00400010   addi $5,$5,-1
0x00400014   j    0x0100001
0x00400018
```

4) Translate to MIPS instructions (write in addrs)

Disassembly Practice Answer (5/9)

```
Address     Instruction
0x00400000   or   $v0,$0,$0
0x00400004   slt  $t0,$0,$a1
0x00400008   beq  $t0,$0,3
0x0040000C   add  $v0,$v0,$a0
0x00400010   addi $a1,$a1,-1
0x00400014   j    0x0100001 # addr: 0x0400004
0x00400018
```

4) Translate to MIPS instructions (write in addrs)

– More readable with register names

Disassembly Practice Answer (6/9)

```
Address     Instruction
0x00400000          or   $v0,$0,$0
0x00400004  Loop:   slt  $t0,$0,$a1
0x00400008          beq  $t0,$0,Exit
0x0040000C          add  $v0,$v0,$a0
0x00400010          addi $a1,$a1,-1
0x00400014          j    Loop
0x00400018  Exit:
```

4) Translate to MIPS instructions (write in addrs)

– Introduce labels

Disassembly Practice Answer (7/9)

Address Instruction

```
       or   $v0,$0,$0    # initialize $v0 to 0
Loop:  slt  $t0,$0,$a1   # $t0 = 0 if 0 >= $a1
       beq  $t0,$0,Exit  # exit if $a1 <= 0
       add  $v0,$v0,$a0  # $v0 += $a0
       addi $a1,$a1,-1   # decrement $a1
       j    Loop
Exit:
```

4) Translate to MIPS instructions (write in addrs)

– What does it do?

Disassembly Practice Answer (8/9)

```
/* a→$v0, b→$a0, c→$a1 */
a = 0;
while(c > 0) {
   a += b;
   c--;
}
```

5) Translate into C code

– Initial direct translation

Disassembly Practice Answer (9/9)

```
/* naïve multiplication: returns m*n */
int multiply(int m, int n) {
  int p; /* product */
  for(p = 0; n-- > 0; p += m) ;
  return p;
}
```

5) Translate into C code

– One of many possible ways to write this