Computer Architectures

Machine Language(2)

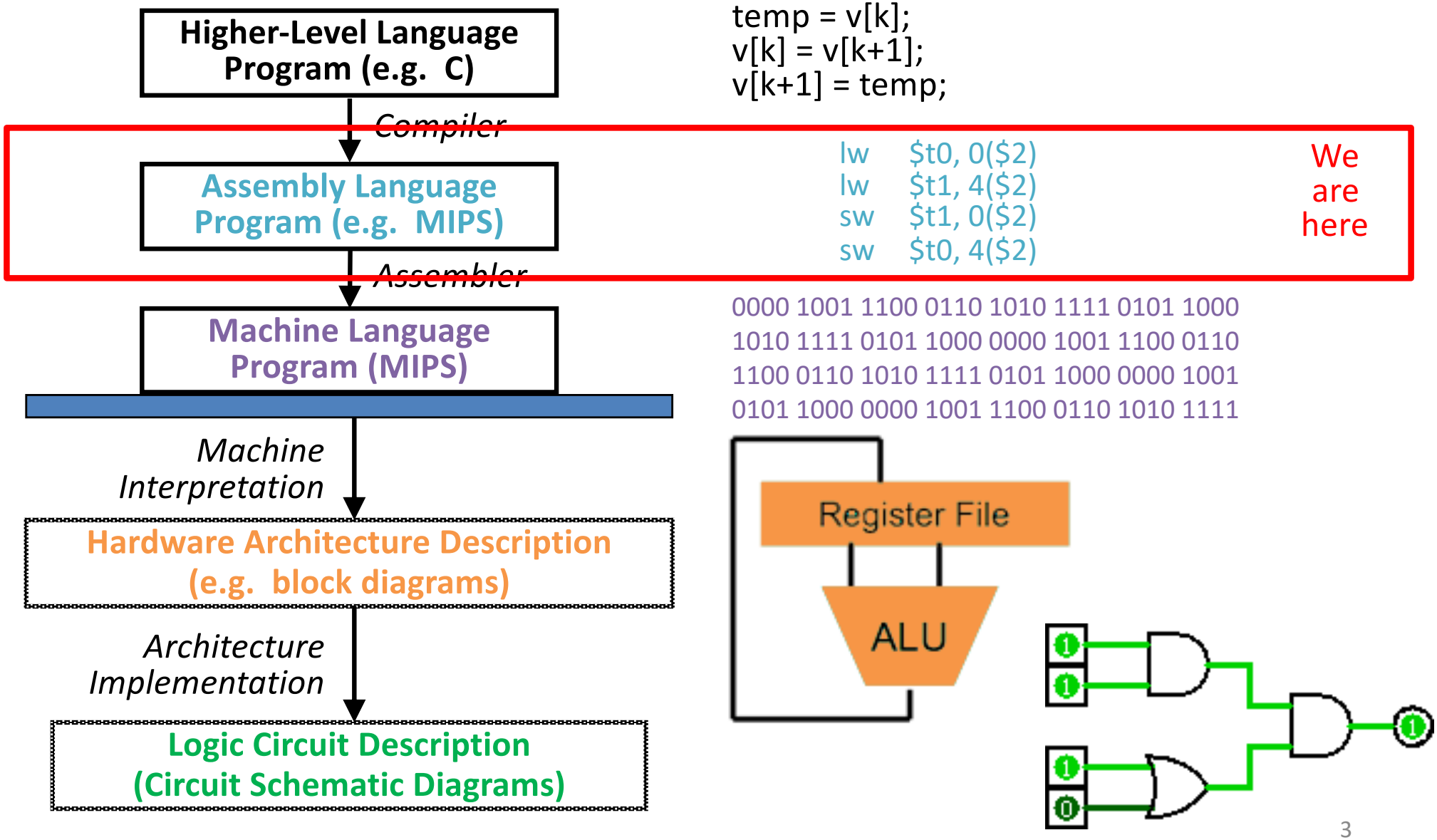
Zhu Wen-Jun

Institute of Computer

Outline

- Stored-Program Concept
- R-Format
- I-Format
  - Branch and PC-Relative Addressing
- J-Format

Great Idea #1: Levels of Representation/Interpretation



Outline

- Stored-Program Concept
- R-Format
- I-Format
  - Branch and PC-Relative Addressing
- J-Format

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

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

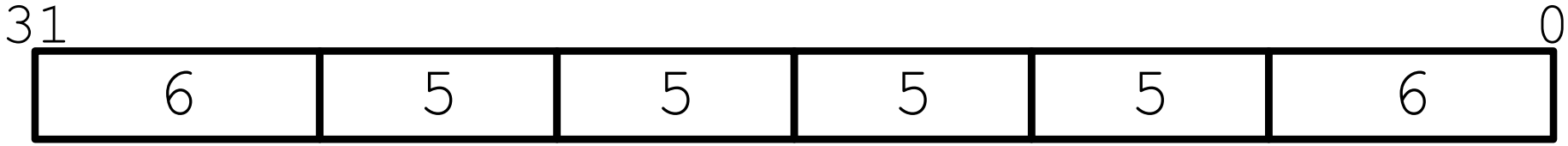
- I-Format: instructions with immediates, `lw`/`sw` (offset is immediate), and `beq`/`bne`
  - But not the shift instructions
- J-Format: `j` and `jal`
  - But not `jr`
- R-Format: all other instructions
- It will soon become clear why the instructions have been partitioned in this way

Outline

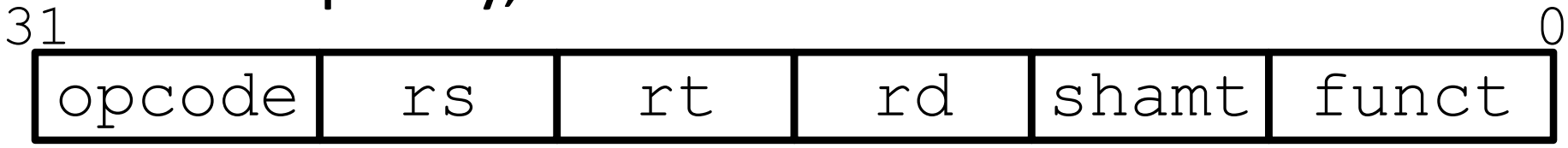
- Stored-Program Concept
- R-Format
- I-Format
  - Branch and PC-Relative Addressing
- J-Format

R-Format Instructions (1/4)

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



- For simplicity, 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

MIPS32® Architecture For Programmers
Volume II: The MIPS32® Instruction Set

Document Number: MD00086
Revision 2.50
July 1, 2005

MIPS Technologies, Inc.
1225 Charleston Road
Mountain View, CA 94043-1353

R-Format Example (1/2)

- MIPS Instruction:

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

- Pseudo-code ("OPERATION" column):

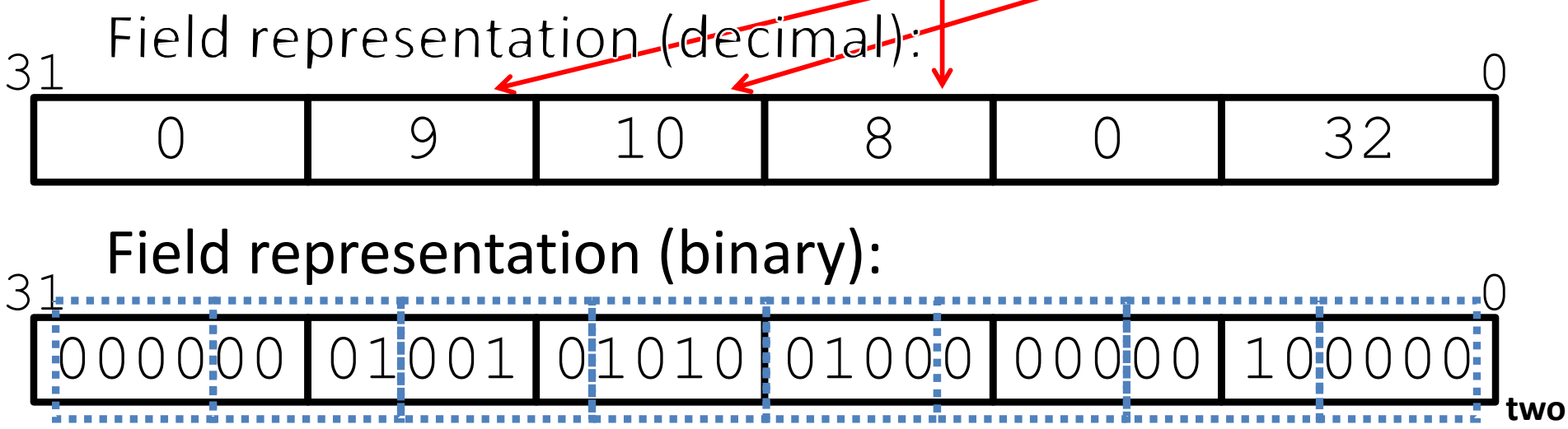
```
add   R[rd] = R[rs] + R[rt]
```

- 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`



hex representation: `0x012A4020`

decimal representation: `19,546,144`

Called a Machine Language Instruction

NOP

- 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` for "No Operation Performed"
  - We'll see its uses later in the course

Outline

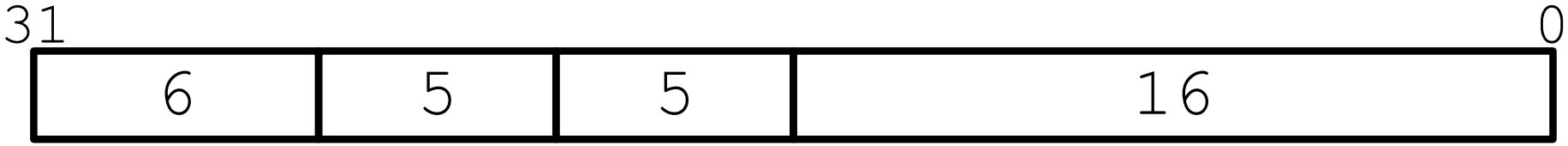
- Stored-Program Concept
- R-Format
- I-Format
  - Branch and PC-Relative Addressing
- J-Format

I-Format Instructions (1/4)

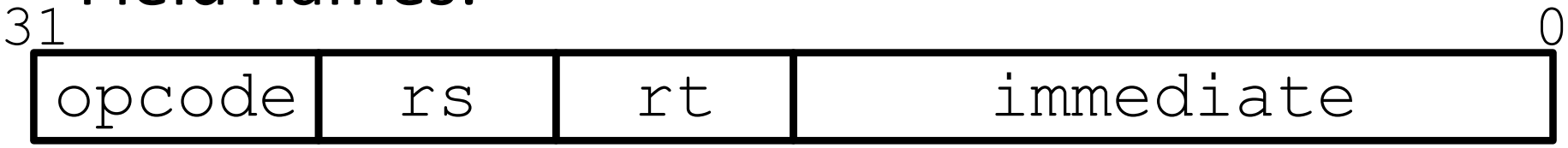
- What about instructions with immediates?
  - 5- and 6-bit fields too small for most immediates
- Ideally, MIPS would have only one instruction format (for simplicity)
  - Unfortunately here we need to compromise
- Define new instruction format that is partially consistent with R-Format
  - First notice 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`
  - We'll see what to do when the number is too big later today…

I-Format Example (1/2)

- MIPS Instruction:

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

- Pseudo-code ("OPERATION" column)

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

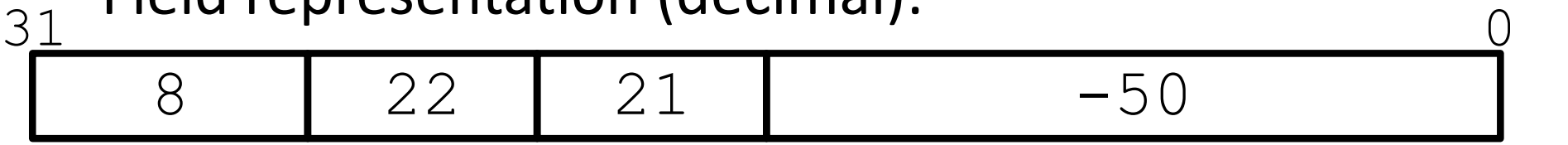
- Fields:

`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)

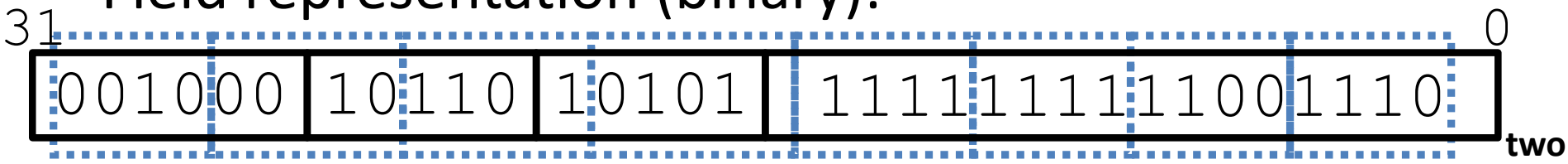
I-Format Example (2/2)

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

Field representation (decimal):



Field representation (binary):



hex representation: `0x22D5FFCE`

decimal representation: `584,449,998`

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

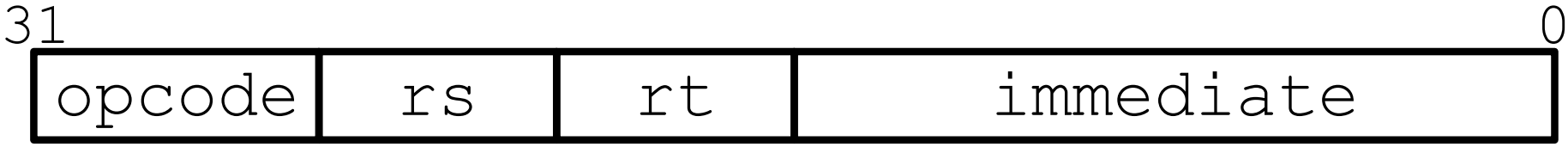
```
lui $at,0xABAB      # upper 16
ori $at,$at,0xCDCD  # lower 16
add $t0,$t0,$at     # move
```

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 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

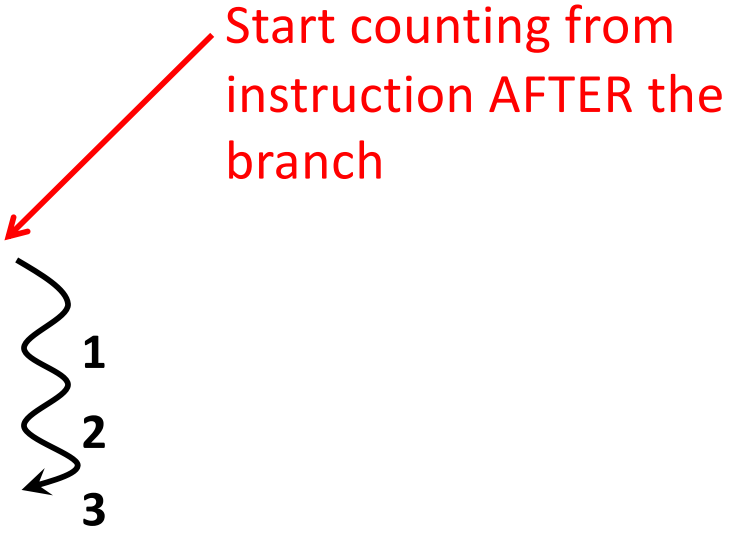
Branch Calculation

- 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; 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)

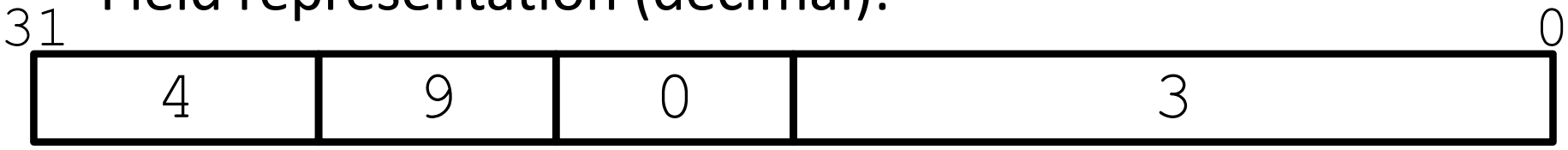
  immediate = 3

Branch Example (2/2)

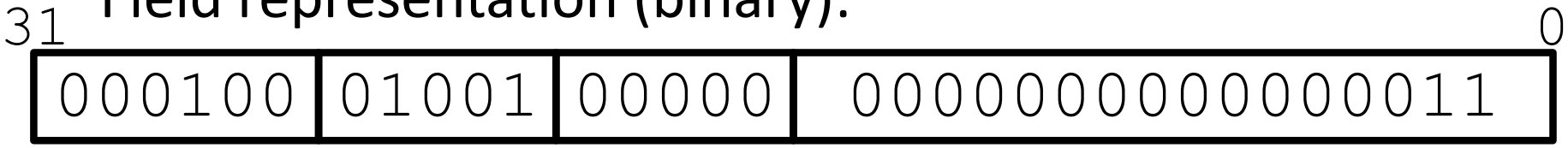
- MIPS Code:

```
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
```

Outline

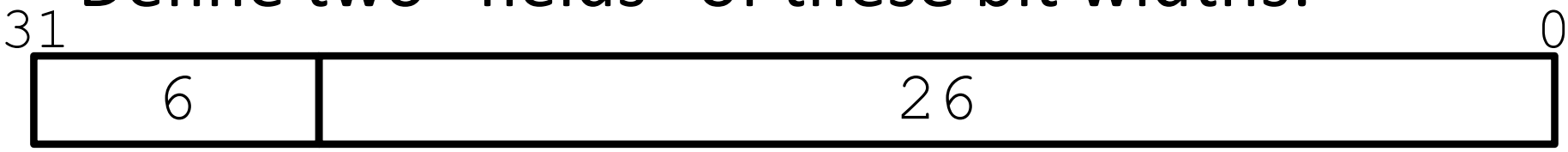
- Stored-Program Concept
- R-Format
- I-Format
  - Branch and PC-Relative Addressing
- J-Format

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)

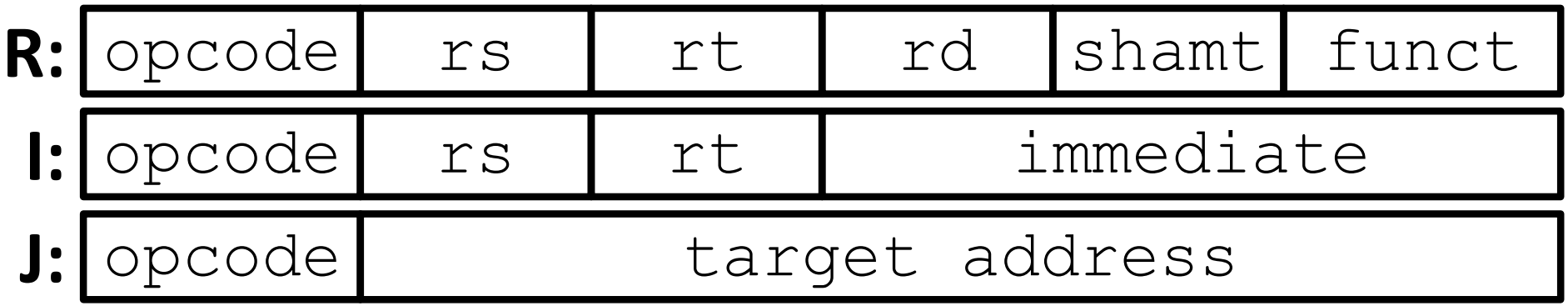
- We can specify 2^{26} addresses
  - Still going to word-aligned instructions, so add `0b00` as last two bits (multiply by 4)
  - This brings us to 28 bits of a 32-bit address
- Take the 4 highest order bits from the PC
  - Cannot reach *everywhere*, but adequate almost all of the time, since programs 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

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

Homework

- 《Computer Organization and Design》
- WORD
  - 2.10，2.11，2.12