# CS 61c: Great Ideas in Computer Architecture

## MIPS Instruction Formats

**Instructor:** Alan Christopher

July 2, 2014

# Review

- New registers: `$a0-$a3`, `$v0-$v1`, `$ra`, `$sp`
- 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 uses and restores before returning

**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));
}
```

---

(blue) We do not need to move the stack at all.
(green) We must save `$ra` on the stack.
(purple) We could copy `$a0` to `$a1` to store `n` across recursive calls.
(yellow) We must save `$a0` on the stack to save it across recursive calls.

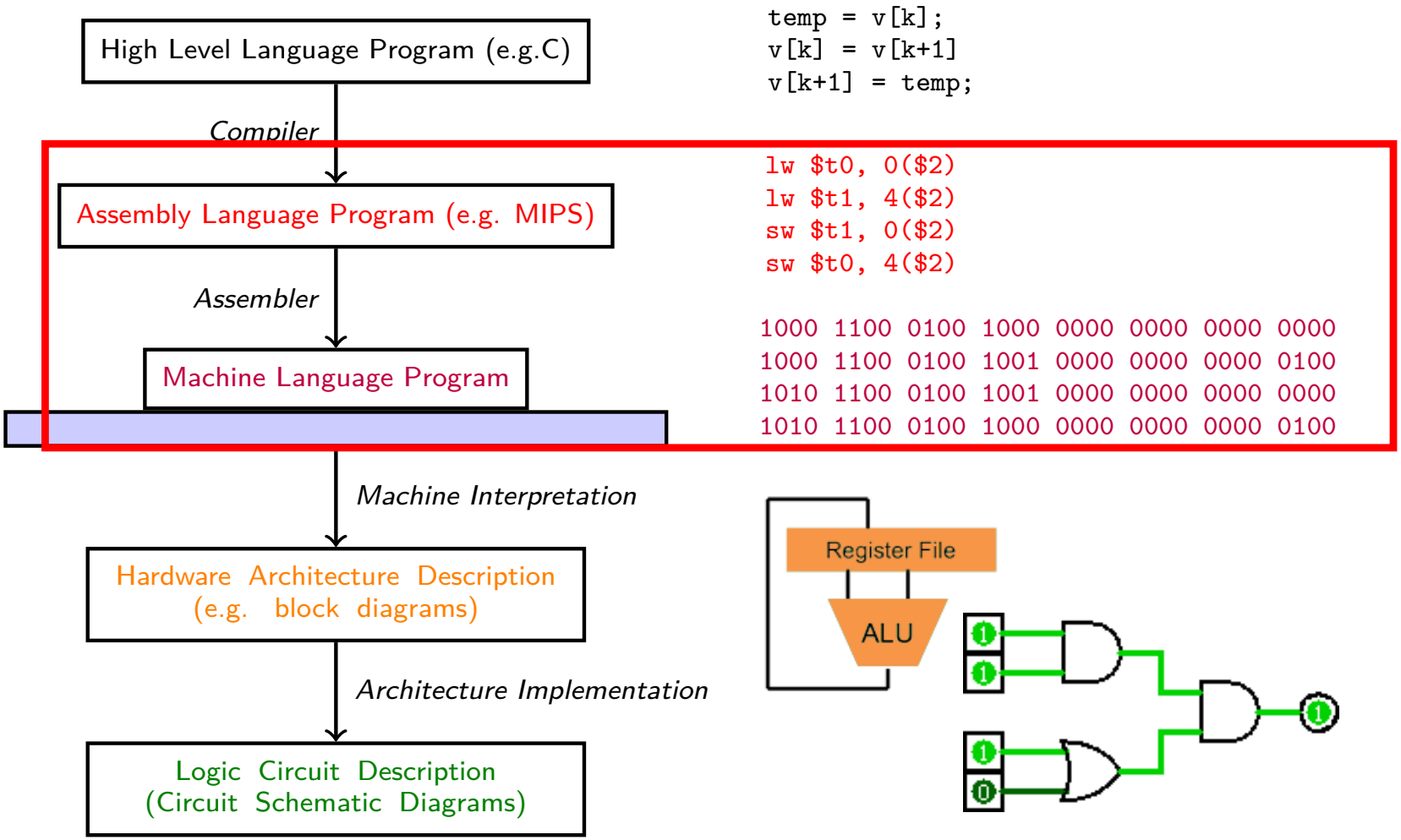
**Question:** Which statement below is **TRUE** about converting the following C code to MIPS?

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

---

(blue) We do not need to move the stack at all.
(green) We must save $ra on the stack.
(purple) We could copy $a0 to $a1 to store n across recursive calls.
(yellow) We must save $a0 on the stack to save it across recursive calls.

# Great Idea #1: Levels of Representation/Interpretation



# Outline

# Big Idea: Stored-Program Concept

- ***Encode instructions as binary data***
  - Instructions can be stored in memory and read/written just like data
- Simplifies SW/HW of computer
  - Same memory technology for instructions and data (because instructions *are* data)
- Stored in memory, so both instructions and data words have addresses
  - A program is just an array of instructions
  - Jump/branch logic is just a special form of pointer arithmetic!

# Binary Compatibility

- Programs are (often) distributed in binary form
  - Programs bound to specific instruction set
  - e.g. 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 x86 in 1981 for 1st IBM PC is a major reason PCs still use x86 instruction set; you could run a program from 1981 PC today.
  - One of the reason x86 is such an ugly language – it's accumulated a lot of cruft.

# Instructions as Bits I

- Currently all data we work with is in words (except chars)
  - All registers are a word wide
  - `lw` and `sw` both access one word of memory
- How do we want to represent instructions?
  - Remember: computer only sees 1s and 0s, so "`add $t0,$t0,$0`" is meaningless
  - KISS: data is in words, put instructions in words too

# Instructions as Bits II

- 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 makes life simpler for the hardware designer, and the ISA designer
- Define 3 types of instruction formats:
  - R-type
  - I-type
  - J-type

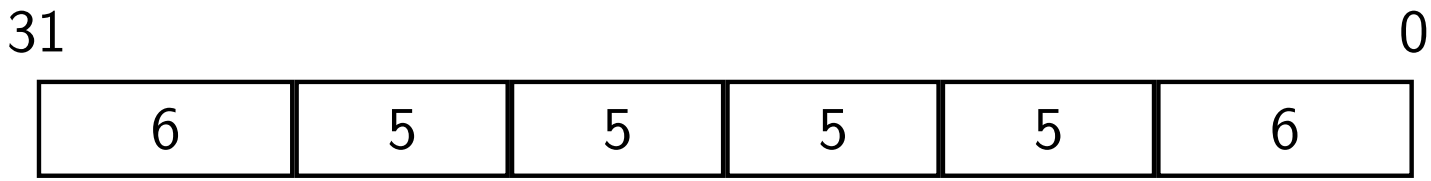
# Instruction formats

- I-type: instructions with immediates, `lw`,`sw` (offset is an immediate), and `beq`/`bne`
  - Does not include shift instructions
- J-type: `j` and `jal`
  - Does not include `jr`
- R-type: Everything else

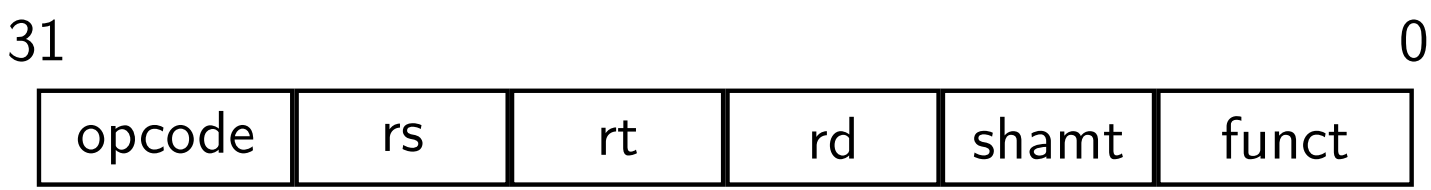
# Outline

# R-Type Instructions I

- Define *fields* of with widths: $6 + 5 + 5 + 5 + 5 + 6 = 32$




- 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-Type Instructions II

- **opcode** (6): Partially specifies operation
  - 0 for all R-type instructions
- **funct** (6): Specifies the instruction, when combined with the opcode
- How many R-type instructions can we encode?

# R-Type Instructions II

- **opcode** (6): Partially specifies operation
  - 0 for all R-type instructions
- **funct** (6): Specifies the instruction, when combined with the opcode
- How many R-type instructions can we encode?
  - Opcode is fixed, so 64
- Why not one 12-bit field?
  - Makes life easier for the other instruction formats, as we'll see.

# R-Type Instructions III

- **rs** (5): specifies the register containing the 1st operand ("source register")
- **rt** (5): specifies the register containing the 2nd operand ("target register")
- **rd** (5): specifies the register which receives the result ("destination register")
- Recall: MIPS has 32 registers
  - Register specifier fits perfectly in 5-bit field
- These map intuitively to instructions
  - e.g. `add dst, src1, src2` → `add rd, rs, rt`
  - Depending on instruction, some fields may not be used

# R-Type Instructions IV

- **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 shift instructions
- Use your Green Sheet for a detailed description of field usage and instruction type for each instruction

# R-Type Example

- MIPS Instruction

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

- Pseudo-code

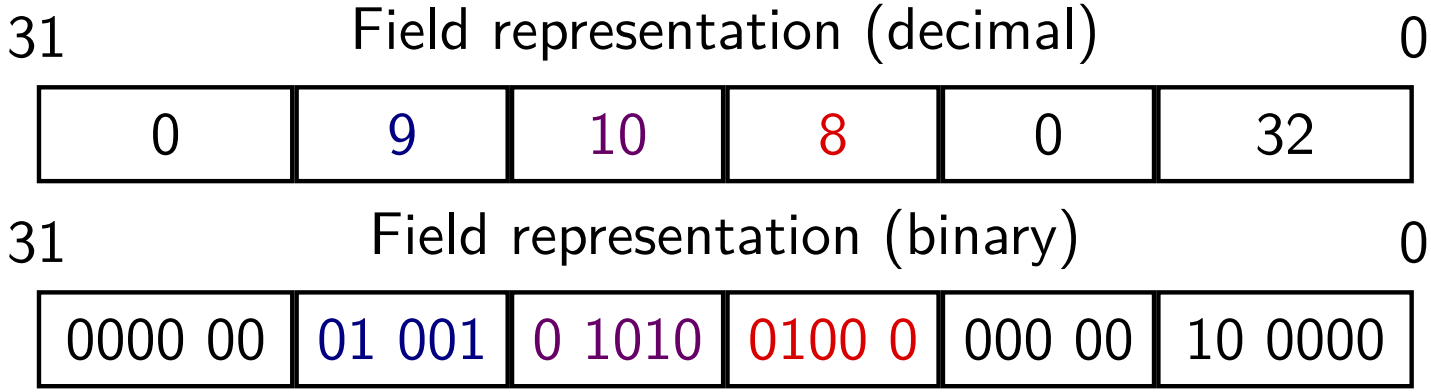
```
R[rd] = R[rs] + R[rt]
```

- Fields:

| | |
|---|---|
| `opcode = 0` | (from Green Sheet) |
| `funct = 32` | (from Green Sheet) |
| `rd = 8` | (destination) |
| `rs = 9` | (first operand) |
| `rt = 10` | (second operand) |
| `shamt = 0` | (not a shift) |

# R-Type Example

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




hex representation: 0x012A4020
decimal representation: 19546144

- This is your first *machine language instruction*

# NOP

- What is the instruction 0x00000000?
  - opcode is 0, so it must be an R-type

# NOP

- What is the instruction 0x00000000?
  - opcode is 0, so it must be an R-type
- Using Green sheet, translates into

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

  - What does this do?

# NOP

- What is the instruction 0x00000000?
  - opcode is 0, so it must be an R-type
- Using Green sheet, translates into

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

  - What does this do? NOTHING!
- This is a special instruction called a `nop` (short for "no operation")

# Outline

# Administrivia

- Midterm exam room and time finalized
  - 5-8pm, 07/21
  - 2050 VLSB
- Proj1/hw2 status check

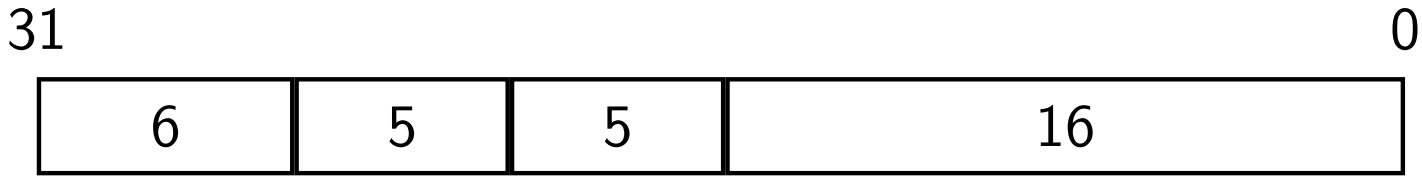
# Outline

# I-Type Instructions I

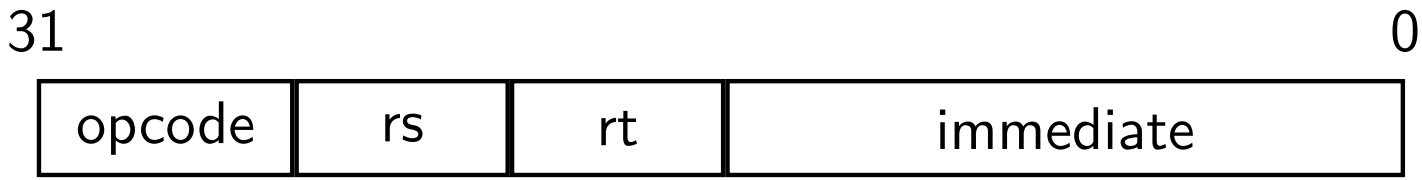
- What about instructions with immediates?
  - 5-bit and 6-bit fields too small for most immediates
- Ideally, MIPS would only have one instruction format (simplicity)
  - But need to support larger immediates, so compromise
- Define a new instruction format which is partially consistent with R-Type

# I-Type Instructions II

- Define fields of with widths: $6 + 5 + 5 + 16 = 32$




- Field names:




- **Important:** The first three fields are consistent with R-Type instructions
  - Most importantly, opcode is still the the same place (this is why we split up funct and opcode)

# I-Type Instructions III

- **opcode** (6): uniquely specifies the instruction
  - All I-Type instructions have non-zero opcode (why?)
- **rs** (5): specifies a register operand
  - Not always used
- **rt** (5): specifies register that receives result of computation ("target register")
  - Name makes more sesnse for I-Type instructions than it did for R-Type

# I-Type Instructions IV

- **immediate** (16):
  - All computations done in words, so 16 bit immediate must be extended to 32 bits
  - Green sheet specifies zeroExtImm or signExtImm based on instruction
  - Usually the "sensible thing" is done, but not always, so it's best to check when in doubt
- Can represent $2^{16}$ different immediates
  - Large enough for the vast majority of constants
  - Load values into a register first when need larger constants

# I-Type Example

- MIPS Instruction:

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

- Pseudo-code

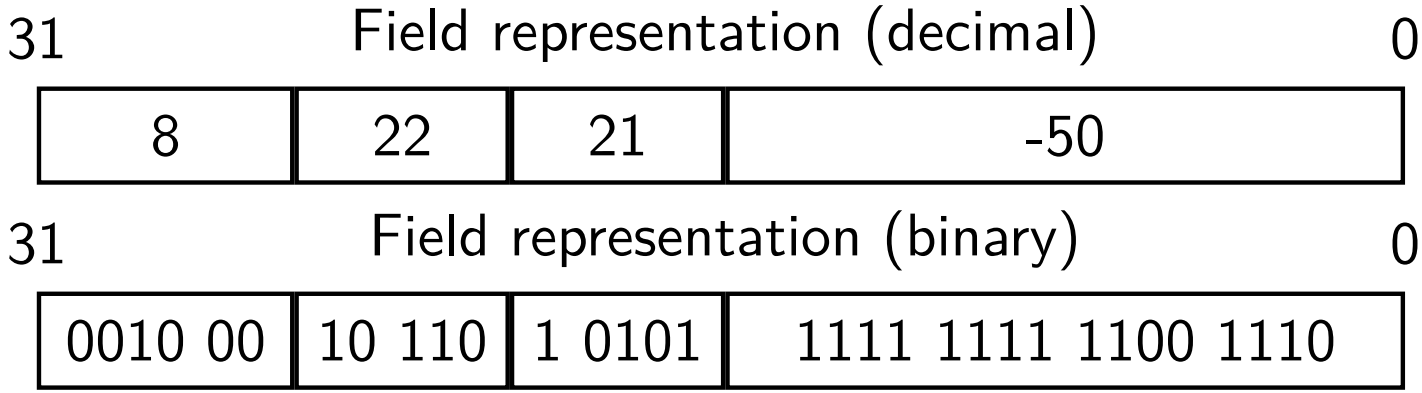
```
R[rt] = R[rs] + SignExtImm
```

- Fields:

| | |
|---|---|
| `opcode = 8` | (from Green Sheet) |
| `rs = 22` | (source register) |
| `rt = 21` | (target register) |
| `imm = -50` | (could also specify in hex) |

# I-Type Example

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




hex representation: 0x22D5FFCE
decimal representation: 584449998

**Question:** Which instruction has the same representation as $35_{10}$?

| Instruction | OPCODE/FUNCT | Register name:numbers | |
|---|---|---|---|
| `subu` | 0/35 | `0:` | `$0` |
| `lw` | 35/-- | `8-15:` | `$t0-$t7` |
| `addi` | 8/-- | `16-23:` | `$s0-$s7` |

(blue) `subu $s0, $s0, $s0`
(green) `lw $0, 0($0)`
(purple) `addi $0, $0, 35`
(yellow) `subu $0,$0,$0`

**Question:** Which instruction has the same representation as $35_{10}$?

| Instruction | OPCODE/FUNCT | Register name:numbers |
|---|---|---|
| `subu` | 0/35 | `0: $0` |
| `lw` | 35/-- | `8-15: $t0-$t7` |
| `addi` | 8/-- | `16-23: $s0-$s7` |

(blue) `subu $s0, $s0, $s0`

(green) `lw $0, 0($0)`

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

(yellow) `subu $0,$0,$0`

# Large Immediates

- How do we deal with 32-bit immediates?
  - Sometimes want to use immediates not in the range $[-2^{15}, 2^{15})$ with `addi`, `lw` ,`sw`, `slti`
  - Bitwise operations with 32-bit numbers
- **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 of `reg` (bits 16-31) and zeros out the lower half (bits 0-15)

# `lui` Example

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

# `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
```

  - Can efficiently handle 32-bit immediates now!
  - Note: only assember should use `$at`

# `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
```

  - Can efficiently handle 32-bit immediates now!
  - Note: only assember should use `$at`
- Could we handle 32 bit immediates without a `lui` instruction?

# `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
```

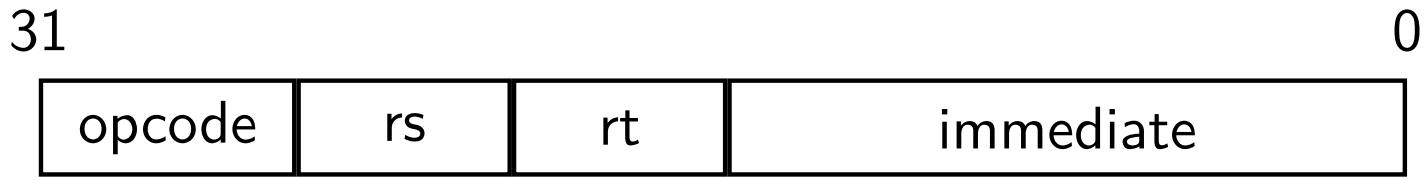
  - Can efficiently handle 32-bit immediates now!
  - Note: only assember should use `$at`
- Could we handle 32 bit immediates without a `lui` instruction?
  - Sure! Break `lui` into `ori` and `sll`:

```
ori $at, $0, 0xABAB
sll $at, $at, 16
```

  - So `lui` doesn't make our code more powerful, but it does make it more efficient.

# Branching Instructions

- `beq` and `bne`
  - Need to specify an address to go to
  - Also take two registers to compare
- Use I-Type:




  - opcode specifies `beq` vs. `bne`
  - `rs` and `rt` specify registers
  - How to 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 usually handled with jump instructions (J-Type)
- **Recall:** Instructions stored in a single segment 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 $\pm 2^{15}$ addresses around the PC
- So just how much memory can we reach?

# Branching Reach

- **Recall:** MIPS uses 32-bit address
  - Memory is byte-addressed
- Instructions are *word-aligned*
  - Address is always a multiple of 4, 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 branch $\pm 2^{15}$ words
  - Can reach $2^{16}$ instructions $= 2^{18}$ bytes around 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)
- **Observe:**
  - immediate is number of *instructions* to jump, either forward (positive), or backwards (negative)
  - Branch from PC+4 for hardware reasons

# Branch Example

- MIPS Code:

```
Loop: beq   $9, $0, End
      addu  $8, $8, $10  # 0
      addiu $9, $9, -1   # 1
      j     Loop         # 2
End:  <some instr>       # 3
```

- I-Type fields:

| | |
|---|---|
| `opcode = 4` | (from Green Sheet) |
| `rs = 9` | (source register) |
| `rt = 0` | (target register) |
| `imm = ???` | |

# Branch Example

- MIPS Code:

```
Loop: beq   $9, $0, End
      addu  $8, $8, $10  # 0
      addiu $9, $9, -1   # 1
      j     Loop         # 2
End:  <some instr>       # 3
```

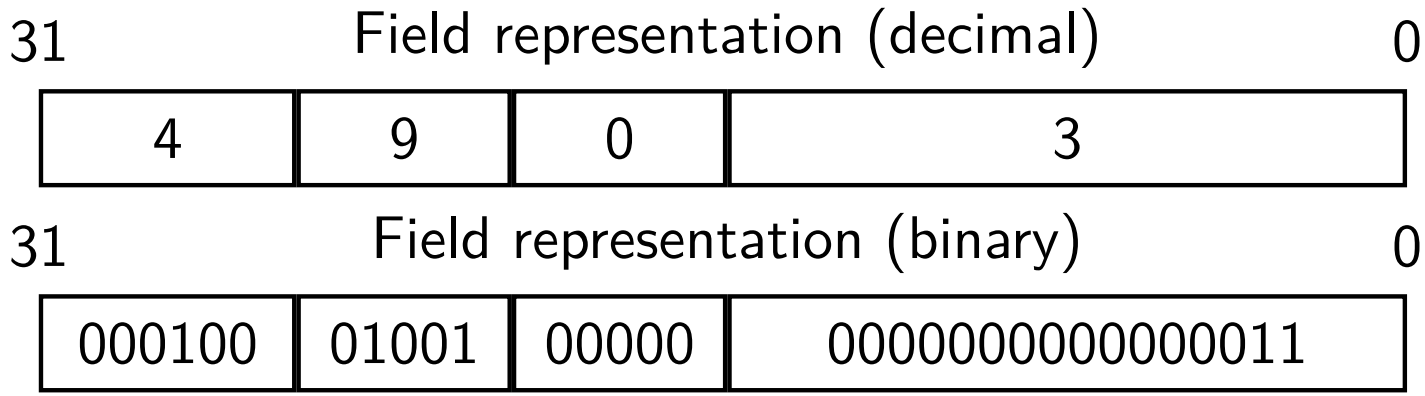
- I-Type fields:

```
opcode = 4   (from Green Sheet)
rs = 9       (source register)
rt = 0       (target register)
imm = 3
```

# Branch Example

- MIPS Code:

```
Loop:  beq    $9, $0, End
       addu   $8, $8, $10  # 0
       addiu  $9, $9, -1   # 1
       j      Loop         # 2
End:   <some  instr>       # 3
```



# Questions on PC-Addressing

- Does the value in branch immediate field change if we move the code?

# 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?
  - Use a jump

```
beq $s0, $0, far
#next inst
```

```
bne $s0, $0, next
j   far
next: #next inst
```

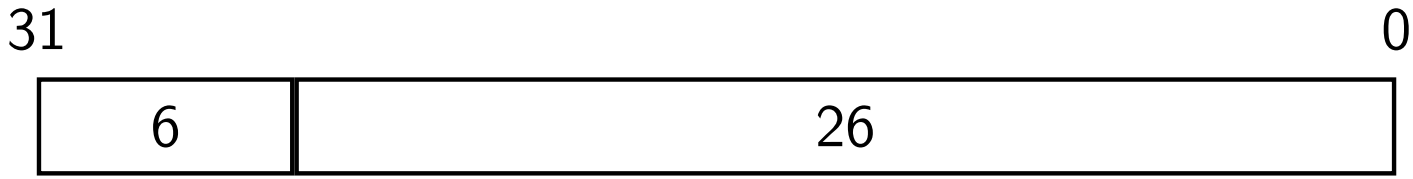
# Technology Break

# J-Type Instructions I

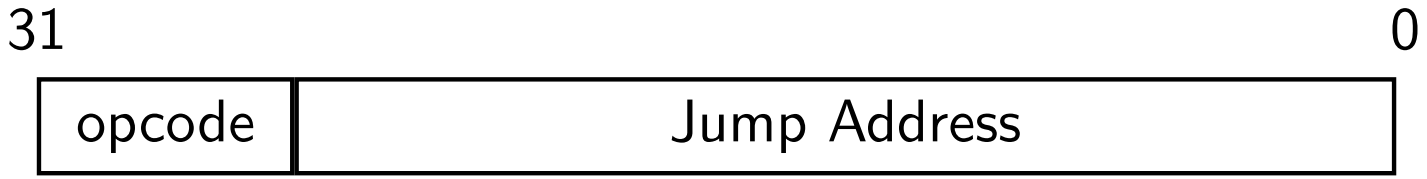
- For branches we assumed that we won't want to branch too far, so we can specify a *change* in the PC
- For true jump instructions (`j` and `jal`) we may want to jump to *anywhere* in memory
  - Ideally, we could specify a 32-bit memory address to which to jump
  - Unfortunately, we can't fit a 32-bit address and a 6-bit opcode into 32 bits

# J-Type Instructions II

- Define fields of with widths: $6 + 26 = 32$




- Field names:




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

# J-Type Instructions III

- We can specify $2^{26}$ addresses
  - Still going to word-aligned instructions, so add 0b00 to the last two bits (i.e. multiply by 4)
  - Still only have 28 bits of 32-bit address

# J-Type Instructions III

- We can specify $2^{26}$ addresses
  - Still going to word-aligned instructions, so add `0b00` to the last two bits (i.e. multiply by 4)
  - Still only have 28 bits of 32-bit address
- Take the 4 highest order bits from the PC
  - Cannot reach *everywhere*, but adequate in almost all circumstances
  - Only a problem if code straddles a 256MiB boudnary
- If necessary, use 2 jumps or `jr` instead

# J-Type Instructions IV

- Jump instruction
  - `New PC = { (PC+4)[31..28], jump address, 0b00 }`
- Note:
  - { , , } means concatenation
  - Array indexing: `[31..28]` means highest 4 bits of PC
  - For hardware reasons, use `PC+4` instead of `PC`

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

**Question:** When merging two or more binaries:

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

| | | |
|---|---|---|
| (blue) | F | F |
| (green) | F | T |
| (purple) | T | F |
| (yellow) | T | T |

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

**Question:** When merging two or more binaries:

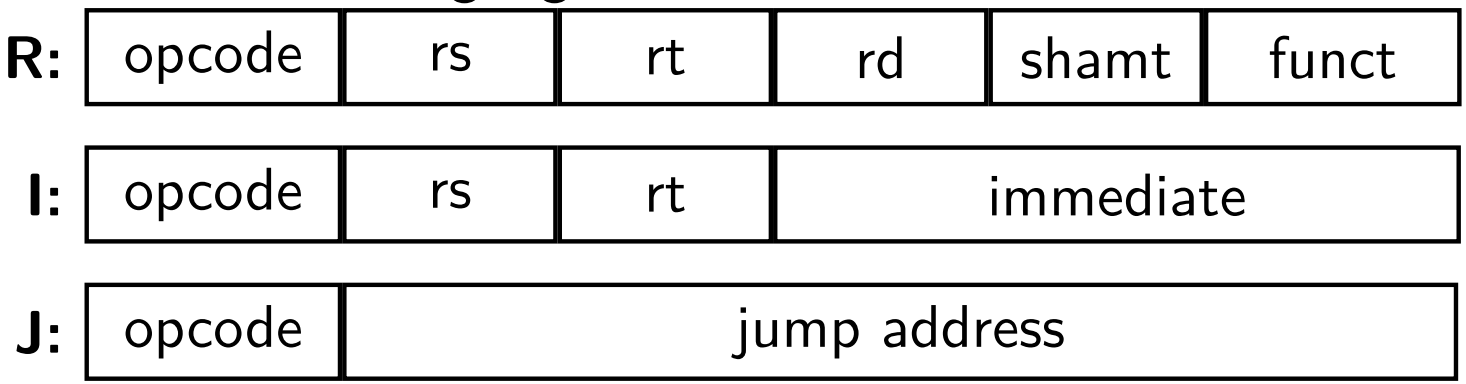
1. Jump instructions don't require any changes
2. Branch instructions don't require any changes

| | | |
|---|---|---|
| (blue) | F | F |
| (green) | F | T |
| (purple) | T | F |
| (yellow) | T | T |

# Outline

# And in Conclusion

- 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 (pseudo)absolute addressing