

ECE-425: Lecture 4

Instruction Set Architecture

Prof: Mohamed El-Hadedy

Email: mealy@cpp.edu

Office: 909-869-2594

MIPS Simulator

- In case of running MIPS simulator
 - <https://www.usna.edu/Users/cs/lmcdowel/courses/si232/spim/PCSPIM-HowTo.htm>
 - Another simulator is called QtSpim
 - Using any editor for writing the program

Logical Operations

- Instructions for bitwise manipulation

| Operation | C | Java | MIPS |
|-------------|----|------|-----------|
| Shift left | << | << | sll |
| Shift right | >> | >>> | srl |
| Bitwise AND | & | & | and, andi |
| Bitwise OR | | | or, ori |
| Bitwise NOT | ~ | ~ | nor |

- Useful for extracting and inserting groups of bits in a word

Shift Operations

| | | | | | |
|--------|--------|--------|--------|--------|--------|
| op | rs | rt | rd | shamt | funct |
| 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |

- shamt: how many positions to shift
- Shift left logical
 - Shift left and fill with 0 bits
 - sll by i bits multiplies by 2^i
- Shift right logical
 - Shift right and fill with 0 bits
 - srl by i bits divides by 2^i (unsigned only)

AND Operations

- Useful to mask bits in a word
 - Select some bits, clear others to 0

and \$t0, \$t1, \$t2

| | |
|------|---|
| \$t2 | 0000 0000 0000 0000 0000 1101 1100 0000 |
| \$t1 | 0000 0000 0000 0000 0011 1100 0000 0000 |
| \$t0 | 0000 0000 0000 0000 0000 1100 0000 0000 |

OR Operations

- Useful to include bits in a word
 - Set some bits to 1, leave others unchanged

or \$t0, \$t1, \$t2

\$t2

| |
|---|
| 0000 0000 0000 0000 0000 1101 1100 0000 |
|---|

\$t1

| |
|---|
| 0000 0000 0000 0000 0011 1100 0000 0000 |
|---|

\$t0

| |
|---|
| 0000 0000 0000 0000 0011 1101 1100 0000 |
|---|

NOT Operations

- Useful to invert bits in a word
 - Change 0 to 1, and 1 to 0
- MIPS has NOR 3-operand instruction
 - $a \text{ NOR } b == \text{NOT} (a \text{ OR } b)$

`nor $t0, $t1, $zero`

Register 0:
always read
as zero



\$t1 0000 0000 0000 0000 0011 1100 0000 0000

\$t0 1111 1111 1111 1111 1100 0011 1111 1111

More Conditional Operations

- Set result to 1 if a condition is true
 - Otherwise, set to 0
- `slt rd, rs, rt`
 - if ($rs < rt$) $rd = 1$; else $rd = 0$;
- `slti rt, rs, constant`
 - if ($rs < \text{constant}$) $rt = 1$; else $rt = 0$;
- Use in combination with `beq`, `bne`
 - `slt $t0, $s1, $s2 # if ($s1 < $s2)`
 - `bne $t0, $zero, L # branch to L`

Branch Instruction Design

- Why not `blt`, `bge`, etc?
- Hardware for `<`, `≥`, ... slower than `=`, `≠`
 - Combining with branch involves more work per instruction, requiring a slower clock
 - All instructions penalized!
- `beq` and `bne` are the common case
- This is a good design compromise

Signed vs. Unsigned

- Signed comparison: `slt`, `slti`
- Unsigned comparison: `sltu`, `sltui`
- Example
 - `$s0 = 1111 1111 1111 1111 1111 1111 1111 1111`
 - `$s1 = 0000 0000 0000 0000 0000 0000 0000 0001`
 - `slt $t0, $s0, $s1 # signed`
 - $-1 < +1 \Rightarrow \$t0 = 1$
 - `sltu $t0, $s0, $s1 # unsigned`
 - $+4,294,967,295 > +1 \Rightarrow \$t0 = 0$

Procedure Calling

- **Steps required**
 1. Place parameters in registers
 2. Transfer control to procedure
 3. Acquire storage for procedure
 4. Perform procedure's operations
 5. Place result in register for caller
 6. Return to place of call

Register Usage

- \$a0 – \$a3: arguments (reg's 4 – 7)
- \$v0, \$v1: result values (reg's 2 and 3)
- \$t0 – \$t9: temporaries
 - Can be overwritten by callee
- \$s0 – \$s7: saved
 - Must be saved/restored by callee
- \$gp: global pointer for static data (reg 28)
- \$sp: stack pointer (reg 29)
- \$fp: frame pointer (reg 30)
- \$ra: return address (reg 31)

Procedure Call Instructions

- Procedure call: jump and link
jal ProcedureLabel
 - Address of following instruction put in \$ra
 - Jumps to target address
- Procedure return: jump register
jr \$ra
 - Copies \$ra to program counter
 - Can also be used for computed jumps
 - e.g., for case/switch statements

Leaf Procedure Example

- C code:

```
int leaf_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

- Arguments g, ..., j in \$a0, ..., \$a3
- f in \$s0 (hence, need to save \$s0 on stack)
- Result in \$v0

Leaf Procedure Example

- MIPS code:

```
leaf_example:  
    addi    $sp,    $sp,    -4  
    sw      $s0,    0($sp)  
    add     $t0,    $a0,    $a1  
    add     $t1,    $a2,    $a3  
    sub     $s0,    $t0,    $t1  
    add     $v0,    $s0,    $zero  
    lw      $s0,    0($sp)  
    addi    $sp,    $sp,    4  
    jr      $ra
```

Byte/Halfword Operations

- Could use bitwise operations
- MIPS byte/halfword load/store
 - String processing is a common case

lb rt, offset(rs) lh rt, offset(rs)

- Sign extend to 32 bits in rt

lbu rt, offset(rs) lhu rt, offset(rs)

- Zero extend to 32 bits in rt

sb rt, offset(rs) sh rt, offset(rs)

- Store just rightmost byte/halfword

String Copy Example

- C code (naïve):

- Null-terminated string

```
void strcpy (char x[], char y[])  
{ int i;  
  i = 0;  
  while ((x[i]=y[i])!='\0')  
    i += 1;  
}
```

- Addresses of x, y in \$a0, \$a1
 - i in \$s0

String Copy Example

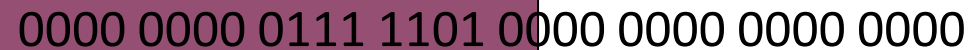
- MIPS code:

```
strcpy:
    addi $sp, $sp, -4      # adjust stack for 1 item
    sw   $s0, 0($sp)      # save $s0
    add  $s0, $zero, $zero # i = 0
L1:    add $t1, $s0, $a1    # addr of y[i] in $t1
        lbu $t2, 0($t1)    # $t2 = y[i]
        add $t3, $s0, $a0  # addr of x[i] in $t3
        sb  $t2, 0($t3)    # x[i] = y[i]
        beq $t2, $zero, L2 # exit loop if y[i] == 0
        addi $s0, $s0, 1   # i = i + 1
        j   L1            # next iteration of loop
L2:    lw   $s0, 0($sp)    # restore saved $s0
        addi $sp, $sp, 4   # pop 1 item from stack
        jr  $ra           # and return
```

32-bit Constants

- Most constants are small
 - 16-bit immediate is sufficient
- For the occasional 32-bit constant
`lui rt, constant`
 - Copies 16-bit constant to left 16 bits of rt
 - Clears right 16 bits of rt to 0

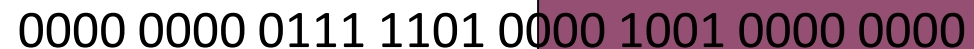
`lui $s0, 125`



0000 0000 0111 1101 0000 0000 0000 0000

A horizontal bar representing a 32-bit register. The first 16 bits (0000 0000 0111 1101) are highlighted in dark red, representing the 16-bit constant 125 shifted left by 16 bits. The remaining 16 bits are 0000 0000 0000 0000.

`ori $s0, $s0, 2304`



0000 0000 0111 1101 0000 1001 0000 0000

A horizontal bar representing a 32-bit register. The first 16 bits (0000 0000 0111 1101) are highlighted in dark red, representing the 16-bit constant 125 shifted left by 16 bits. The next 16 bits (0000 1001 0000 0000) are highlighted in dark red, representing the 16-bit constant 2304. The final 16 bits are 0000 0000 0000 0000.

Branch Addressing

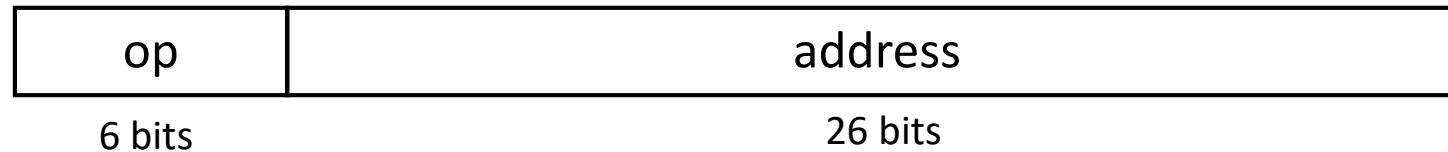
- Branch instructions specify
 - Opcode, two registers, target address
- Most branch targets are near branch
 - Forward or backward



- PC-relative addressing
 - Target address = $PC + \text{offset} \times 4$
 - PC already incremented by 4 by this time

Jump Addressing

- Jump (j and jal) targets could be anywhere in text segment
 - Encode full address in instruction



- (Pseudo)Direct jump addressing
 - Target address = $PC_{31...28} : (\text{address} \times 4)$

Branching Far Away

- If branch target is too far to encode with 16-bit offset, assembler rewrites the code
- Example

```
        beq $s0,$s1, L1
           ↓
        bne $s0,$s1, L2
j L1
L2:    ...
```

Addressing Mode Summary

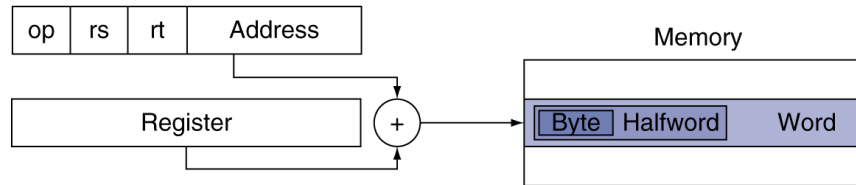
1. Immediate addressing



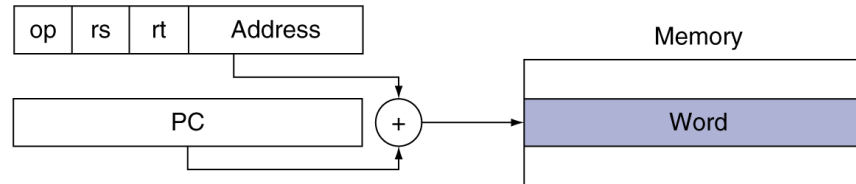
2. Register addressing



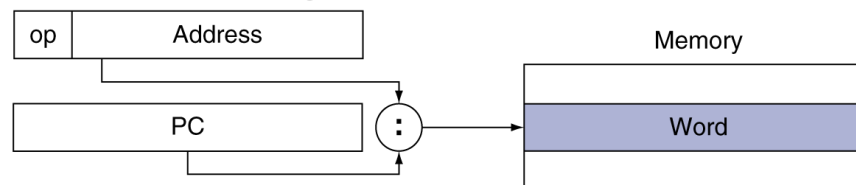
3. Base addressing



4. PC-relative addressing



5. Pseudodirect addressing



Summery

Instruction

add \$s1,\$s2,\$s3
sub \$s1,\$s2,\$s3
lw \$s1,100(\$s2)
sw \$s1,100(\$s2)
bne \$s4,\$s5,L
beq \$s4,\$s5,L
j Label

Meaning

\$s1 = \$s2 + \$s3
\$s1 = \$s2 - \$s3
\$s1 = Memory[\$s2+100]
Memory[\$s2+100] = \$s1
Next instr. is at Label if \$s4 = \$s5
Next instr. is at Label if \$s4 ≠ \$s5
Next instr. is at Label

Formats:

| | | | | | | |
|---|----|----------------|----|----------------|-------|-------|
| R | op | rs | rt | rd | shamt | funct |
| I | op | rs | rt | 16 bit address | | |
| J | op | 26 bit address | | | | |

Registers

| Name | Register number | Usage |
|-----------|-----------------|--|
| \$zero | 0 | the constant value 0 |
| \$v0-\$v1 | 2-3 | values for results and expression evaluation |
| \$a0-\$a3 | 4-7 | arguments |
| \$t0-\$t7 | 8-15 | temporaries |
| \$s0-\$s7 | 16-23 | saved |
| \$t8-\$t9 | 24-25 | more temporaries |
| \$gp | 28 | global pointer |
| \$sp | 29 | stack pointer |
| \$fp | 30 | frame pointer |
| \$ra | 31 | return address |

Register 1, \$at, is reserved for the assembler use
Registers 26&27, \$k0&\$k1, reserved by the O.S.

Overview of MIPS

- Simple instructions - all 32 bits wide
- Very structured
- Only three instruction formats

| | | | | | | |
|---|----|----------------|----|----------------|-------|-------|
| R | op | rs | rt | rd | shamt | funct |
| I | op | rs | rt | 16 bit address | | |
| J | op | 26 bit address | | | | |

- Rely on compiler to achieve performance
— what are the compiler's goals?

MIPS Addressing Modes Summary

- Register addressing, operands in registers
- Base or displacement addressing, eff addr. is the sum of register and a constant in inst.
- Immediate addressing, operand is constant within instruction
- PC-relative addressing, $PC + \text{constant}$ in instruction
- Pseudodirect addressing, eff. address is 26 bits in instr. and the upper 4 bits of PC

MIPS assembly language

| Category | Instruction | Example | Meaning | Comments |
|--------------------|-------------------------|----------------------|---|-----------------------------------|
| Arithmetic | add | add \$s1, \$s2, \$s3 | $\$s1 = \$s2 + \$s3$ | Three operands; data in registers |
| | subtract | sub \$s1, \$s2, \$s3 | $\$s1 = \$s2 - \$s3$ | Three operands; data in registers |
| | add immediate | addi \$s1, \$s2, 100 | $\$s1 = \$s2 + 100$ | Used to add constants |
| Data transfer | load word | lw \$s1, 100(\$s2) | $\$s1 = \text{Memory}[\$s2 + 100]$ | Word from memory to register |
| | store word | sw \$s1, 100(\$s2) | $\text{Memory}[\$s2 + 100] = \$s1$ | Word from register to memory |
| | load byte | lb \$s1, 100(\$s2) | $\$s1 = \text{Memory}[\$s2 + 100]$ | Byte from memory to register |
| | store byte | sb \$s1, 100(\$s2) | $\text{Memory}[\$s2 + 100] = \$s1$ | Byte from register to memory |
| | load upper immediate | lui \$s1, 100 | $\$s1 = 100 * 2^{16}$ | Loads constant in upper 16 bits |
| Conditional branch | branch on equal | beq \$s1, \$s2, 25 | if ($\$s1 == \$s2$) go to PC + 4 + 100 | Equal test; PC-relative branch |
| | branch on not equal | bne \$s1, \$s2, 25 | if ($\$s1 \neq \$s2$) go to PC + 4 + 100 | Not equal test; PC-relative |
| | set on less than | slt \$s1, \$s2, \$s3 | if ($\$s2 < \$s3$) $\$s1 = 1$; else $\$s1 = 0$ | Compare less than; for beq, bne |
| | set less than immediate | slti \$s1, \$s2, 100 | if ($\$s2 < 100$) $\$s1 = 1$; else $\$s1 = 0$ | Compare less than constant |
| Unconditional jump | jump | j 2500 | go to 10000 | Jump to target address |
| | jump register | jr \$ra | go to \$ra | For switch, procedure return |
| | jump and link | jal 2500 | $\$ra = PC + 4$; go to 10000 | For procedure call |

ARM & MIPS Similarities

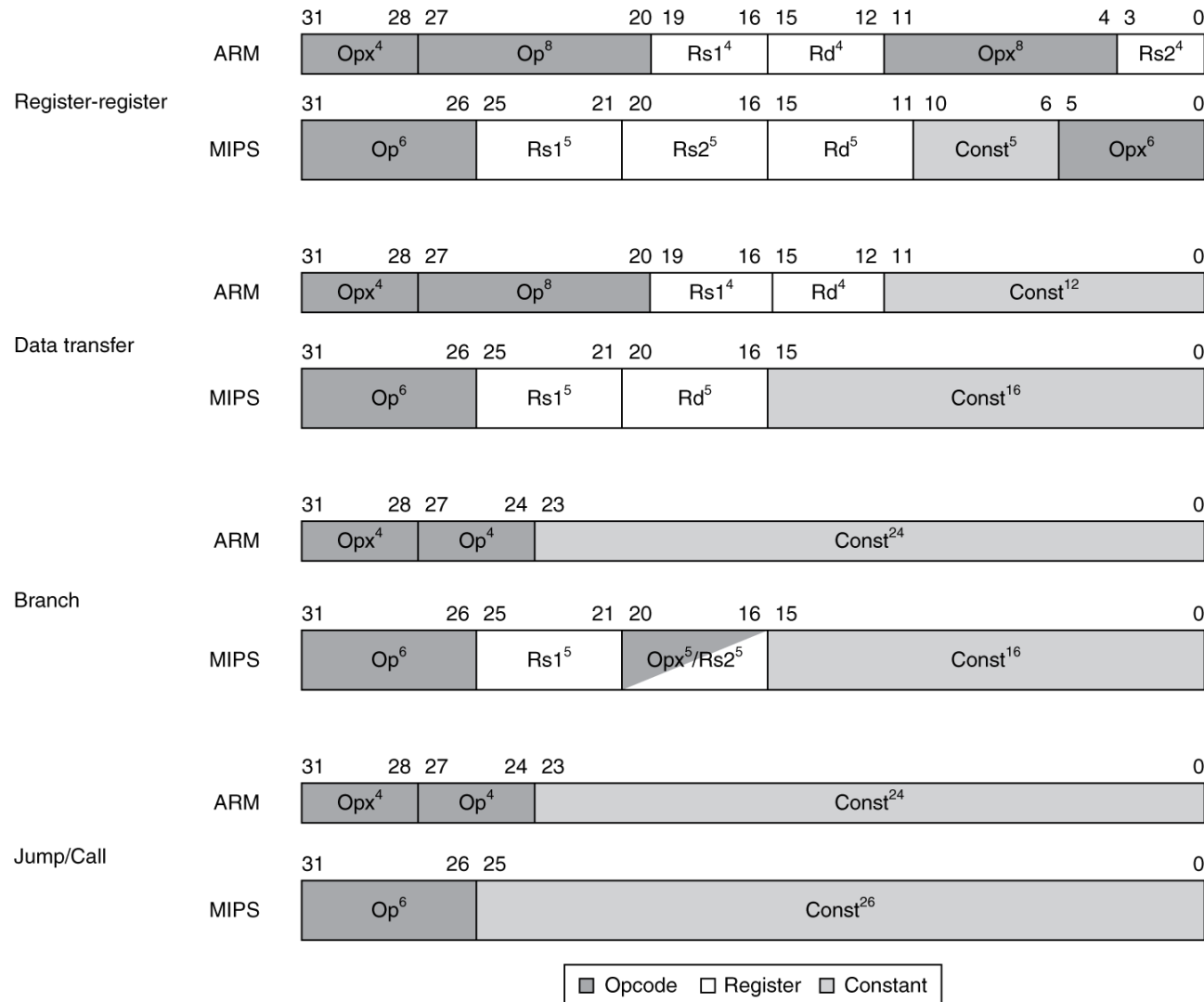
- ARM: the most popular embedded core
- Similar basic set of instructions to MIPS

| | ARM | MIPS |
|-----------------------|---------------|---------------|
| Date announced | 1985 | 1985 |
| Instruction size | 32 bits | 32 bits |
| Address space | 32-bit flat | 32-bit flat |
| Data alignment | Aligned | Aligned |
| Data addressing modes | 9 | 3 |
| Registers | 15 × 32-bit | 31 × 32-bit |
| Input/output | Memory mapped | Memory mapped |

Compare and Branch in ARM

- Uses condition codes for result of an arithmetic/logical instruction
 - Negative, zero, carry, overflow
 - Compare instructions to set condition codes without keeping the result
- Each instruction can be conditional
 - Top 4 bits of instruction word: condition value
 - Can avoid branches over single instructions

Instruction Encoding



MIPS Summary

- Fixed instruction format (3 formats: R, I, J)
- 3-address, reg-reg arithmetic/logical instructions
- Single addressing mode for load/store instructions
 - base + displacement
- Simple branch conditions
 - compare against zero or two registers for equal, not_equal, no integer condition code
- Delayed branch
 - execute instruction after the branch (or jump) even if the branch is not taken. compiler fills a delayed branch with a useful instruction (50% correct).
- Three design principles:
 - Simplicity favors regularity
 - Smaller is faster
 - Good design demands good compromises

MIPS Summary

- Fixed instruction format (3 formats: R, I, J)
- 3-address, reg-reg arithmetic/logical instructions
- Single addressing mode for load/store instructions
 - base + displacement
- Simple branch conditions
 - compare against zero or two registers for equal, not_equal, no integer condition code
- Delayed branch
 - execute instruction after the branch (or jump) even if the branch is not taken. compiler fills a delayed branch with a useful instruction (50% correct).
- Three design principles:
 - Simplicity favors regularity
 - Smaller is faster
 - Good design demands good compromises

Reference:

- **Book:** D. A. Patterson and J. L. Hennessy, **Computer Organization and Design: The Hardware/ Software Interface**, 5th Edition, San Mateo, CA: Morgan and Kaufmann. ISBN: 1-55860-604-1
- **Dr. El-Naga:** ECE-425 Material
- <https://www.mips.com/>
- <https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/mips/index.html>
- **Simulator:** <http://spimsimulator.sourceforge.net/>
- <https://www.usna.edu/Users/cs/lmcdowel/courses/si232/spim/PCSPIM-HowTo.htm>
-