MIPS Assembly

E155

Outline

- MIPS Architecture
- ISA
  - Instruction types
  - Machine codes
- Procedure call
- Stack

The MIPS Register Set

Name	Register Number	Usage
$0	0	the constant value 0
$at	1	assembler temporary
$v0-$v1	2-3	procedure return values
$a0-$a3	4-7	procedure arguments
$t0-$t7	8-15	temporaries
$s0-$s7	16-23	saved variables
$t8-$t9	24-25	more temporaries
$k0-$k1	26-27	OS temporaries
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	procedure return address

R-Type

- *Register-type*
- 3 register operands:
  - `rs`, `rt`: source registers
  - `rd`: destination register
- Other fields:
  - `op`: the *operation code* or *opcode* (0 for R-type instructions)
  - `funct`: the *function*

    together, the opcode and function tell the computer what operation to perform
  - `shamt`: the *shift amount* for shift instructions, otherwise it's 0

I-Type

- *Immediate-type*
- 3 operands:
  - `rs`, `rt`: register operands
  - `imm`: 16-bit two's complement immediate
- Other fields:
  - `op`: the opcode
  - Simplicity favors regularity: all instructions have opcode
  - Operation is completely determined by the opcode

I-Type

Machine Language: J-Type

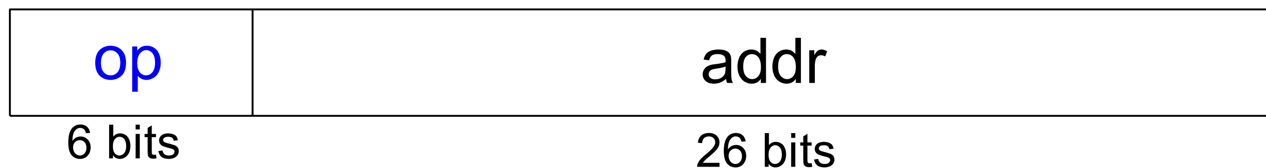
- *Jump-type*
- 26-bit address operand (`addr`)
- Used for jump instructions (`j`)

Review: Instruction Formats

R-Type

I-Type

J-Type



Logical Instructions

- `and`, `or`, `xor`, `nor`
  - `and`: useful for *masking* bits
    - Masking all but the least significant byte of a value:
      0xF234012F AND 0x000000FF = 0x0000002F
  - `or:` useful for combining bit fields
    - Combine 0xF2340000 with 0x000012BC:
      0xF2340000 OR 0x000012BC = 0xF23412BC
  - `nor:` useful for inverting bits:
    - A NOR $0 = NOT A
- `andi`, `ori`, `xori`
  - 16-bit immediate is zero-extended (*not* sign-extended)
  - `nori` not needed

Logical Instruction Examples

Source Registers

$s1	1111	1111	1111	1111	0000	0000	0000	0000
$s2	0100	0110	1010	0001	1111	0000	1011	0111

Assembly Code

```
and $s3, $s1, $s2
or  $s4, $s1, $s2
xor $s5, $s1, $s2
nor $s6, $s1, $s2
```

Result

$s3	0100	0110	1010	0001	0000	0000	0000	0000
$s4	1111	1111	1111	1111	1111	0000	1011	0111
$s5	1011	1001	0101	1110	1111	0000	1011	0111
$s6	0000	0000	0000	0000	0000	1111	0100	1000

Shift Instructions

- `sll:` shift left logical
  - **Example:** `sll $t0, $t1, 5  # $t0 <= $t1 << 5`
- `srl:` shift right logical
  - **Example:** `srl $t0, $t1, 5  # $t0 <= $t1 >> 5`
- `sra:` shift right arithmetic
  - **Example:** `sra $t0, $t1, 5  # $t0 <= $t1 >>> 5`

Variable shift instructions:

- `sllv:` shift left logical variable
  - **Example:** `sllv $t0, $t1, $t2 # $t0 <= $t1 << $t2`
- `srlv:` shift right logical variable
  - **Example:** `srlv $t0, $t1, $t2 # $t0 <= $t1 >> $t2`
- `srav:` shift right arithmetic variable
  - **Example**: `srav $t0, $t1, $t2 # $t0 <= $t1 >>> $t2`

Shift Instructions

Assembly Code / Field Values

Assembly Code	op	rs	rt	rd	shamt	funct
`sll $t0, $s1, 2`	0	0	17	8	2	0
`srl $s2, $s1, 2`	0	0	17	18	2	2
`sra $s3, $s1, 2`	0	0	17	19	2	3
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Machine Code

op	rs	rt	rd	shamt	funct	
000000	00000	10001	01000	00010	000000	(0x00114080)
000000	00000	10001	10010	00010	000010	(0x00119082)
000000	00000	10001	10011	00010	000011	(0x00119883)
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

Generating Constants

- 16-bit constants using `addi`:

High-level code

```
// int is a 32-bit signed word
int a = 0x4f3c;
```

MIPS assembly code

```
# $s0 = a
addi $s0, $0, 0x4f3c
```

- 32-bit constants using load upper immediate (`lui`) and `ori`:

  (`lui` loads the 16-bit immediate into the upper half of the register and sets the lower half to 0.)

High-level code

```
int a = 0xFEDC8765;
```

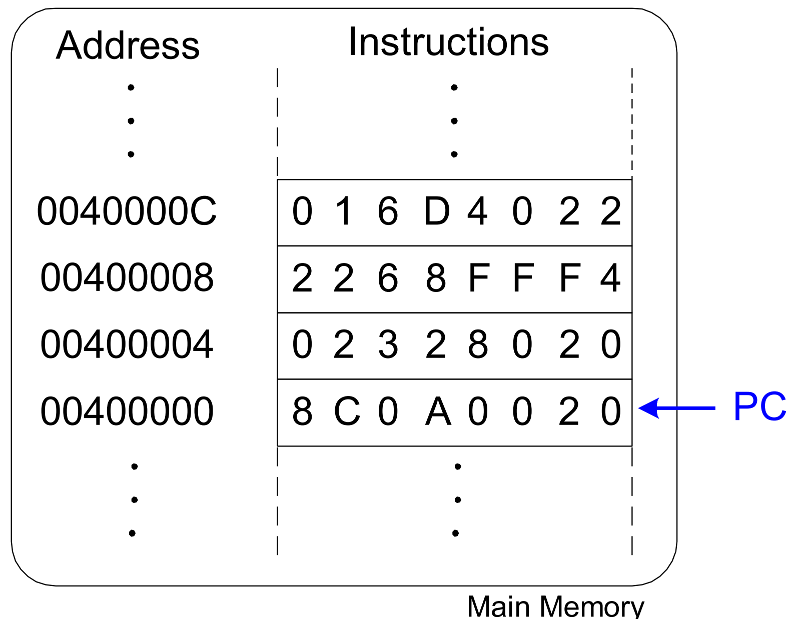
MIPS assembly code

```
# $s0 = a
lui $s0, 0xFEDC
ori $s0, $s0, 0x8765
```

The Stored Program

Assembly Code	Machine Code
`lw   $t2, 32($0)`	`0x8C0A0020`
`add  $s0, $s1, $s2`	`0x02328020`
`addi $t0, $s3, -12`	`0x2268FFF4`
`sub  $t0, $t3, $t5`	`0x016D4022`

Stored Program



Branching

- Allows a program to execute instructions out of sequence.
- Types of branches:
  - **Conditional branches**
    - branch if equal (`beq`)
    - branch if not equal (`bne`)
  - **Unconditional branches**
    - jump (`j`)
    - jump register (`jr`)
    - jump and link (`jal`)

Conditional Branching (beq)

```
# MIPS assembly
 addi $s0, $0, 4              # $s0 = 0 + 4 = 4
 addi $s1, $0, 1              # $s1 = 0 + 1 = 1
 sll  $s1, $s1, 2             # $s1 = 1 << 2 = 4
 beq  $s0, $s1, target        # branch is taken
 addi $s1, $s1, 1             # not executed
 sub  $s1, $s1, $s0           # not executed

target:                       # label
 add  $s1, $s1, $s0           # $s1 = 4 + 4 = 8
```

Labels indicate instruction locations in a program. They cannot use reserved words and must be followed by a colon (:).

Branch Not Taken

```
# MIPS assembly
  addi    $s0, $0, 4         # $s0 = 0 + 4 = 4
  addi    $s1, $0, 1         # $s1 = 0 + 1 = 1
  sll     $s1, $s1, 2        # $s1 = 1 << 2 = 4
  bne     $s0, $s1, target   # branch not taken
  addi    $s1, $s1, 1        # $s1 = 4 + 1 = 5
  sub     $s1, $s1, $s0      # $s1 = 5 – 4 = 1

target:
  add     $s1, $s1, $s0      # $s1 = 1 + 4 = 5
```

Unconditional Branching / Jumping (j)

```
# MIPS assembly
  addi $s0, $0, 4              # $s0 = 4
   addi $s1, $0, 1             # $s1 = 1
   j         target            # jump to target
   sra       $s1, $s1, 2       # not executed
   addi      $s1, $s1, 1       # not executed
   sub       $s1, $s1, $s0     # not executed

  target:
   add       $s1, $s1, $s0     # $s1 = 1 + 4 = 5
```

Procedure Call

MIPS assembly code

```
# $s0 = y

main:
  ...
  addi $a0, $0, 2     # argument 0 = 2
  addi $a1, $0, 3     # argument 1 = 3
  addi $a2, $0, 4     # argument 2 = 4
  addi $a3, $0, 5     # argument 3 = 5
  jal  diffofsums     # call procedure
  add  $s0, $v0, $0   # y = returned value
  ...
                      # jal: jumps to diffofsums  and saves PC+4 in the
                      # return address register ($ra).

# $s0 = result        # jr $ra: jumps to address in $ra.
diffofsums:
  add $t0, $a0, $a1   # $t0 = f + g
  add $t1, $a2, $a3   # $t1 = h + i
  sub $s0, $t0, $t1   # result = (f + g) - (h + i)
  add $v0, $s0, $0    # put return value in $v0
  jr  $ra             # return to caller
```

Stack

```
# $s0 = result
diffofsums:
  addi $sp, $sp, -12   # make space on stack
                       # to store 3 registers
  sw   $s0, 8($sp)     # save $s0 on stack
  sw   $t0, 4($sp)     # save $t0 on stack
  sw   $t1, 0($sp)     # save $t1 on stack
  add  $t0, $a0, $a1   # $t0 = f + g
  add  $t1, $a2, $a3   # $t1 = h + i
  sub  $s0, $t0, $t1   # result = (f + g) - (h + i)
  add  $v0, $s0, $0    # put return value in $v0
  lw   $t1, 0($sp)     # restore $t1 from stack
  lw   $t0, 4($sp)     # restore $t0 from stack
  lw   $s0, 8($sp)     # restore $s0 from stack
  addi $sp, $sp, 12    # deallocate stack space
  jr   $ra             # return to caller
```

Procedure Call

- Caller
  - Put arguments in `$a0-$a3`
  - Save any registers that are needed (`$ra`, maybe `$t0-t9`)
  - `jal callee`
  - Restore registers
  - Look for result in `$v0`
- Callee
  - Save registers that might be disturbed (`$s0-$s7`)
  - Perform procedure
  - Put result in `$v0`
  - Restore registers
  - `jr $ra`