MIPS Introduction

Philipp Koehn
presented by Chang Hwan Choi

~~12 March 2018~~

25 Sept 2019

HW3 due Friday 9/27

Midterm - Friday 10/4
- review materials
  soon
- covers digital logic,
  SCRAM, 6502
  (not MIPS)

RISC

- Developed by MIPS Technologies in 1984, first product in 1986

- Used in

  - Silicon Graphics (SGI) Unix workstations
  - Digital Equipment Corporation (DEC) Unix workstation
  - Nintendo 64
  - Sony PlayStation

- Inspiration for ARM (esp. v8)

Overview

- 32 bit architecture (registers, memory addresses)

$$2^{32} = 4G$$

- 32 registers

- Similar types of instructions to 6502

- Multiply and divide instructions

- Floating point numbers

Example: Addition

- Mathematical view of addition

$$a = b + c$$

Example: Addition

- Mathematical view of addition

$$a = b + c$$

- MIPS instruction

maybe same

add a,b,c

could be same

  a, b, c are registers

- Some are special

```
 0  $zero   always has the value 0
31  $ra     contains return address
```

32 Registers

- Some are special

```
 0   $zero   always has the value 0
31   $ra     contains return address
```

- Some have usage conventions

```
        1    $at    reserved for pseudo-instructions
```

32 Registers

- Some are special

```
 0   $zero   always has the value 0
31   $ra     contains return address
```

- Some have usage conventions

```
  1     $at     reserved for pseudo-instructions
2-3   $v0-$v1   return values of a function call
4-7   $a0-$a3   arguments for a function call
```

- Some are special

```
   0   $zero   always has the value 0
  31   $ra     contains return address
```

"Caller Saved"

- Some have usage conventions

```
         1      $at      reserved for pseudo-instructions
       2-3   $v0-$v1   return values of a function call
       4-7   $a0-$a3   arguments for a function call
 8-15,24,25  $t0-$t9   temporaries, can be overwritten by function
     16-23   $s0-$s7   saved, have to be preserved by function
```

"callee-saved"

- Some are special

```
 0   $zero   always has the value 0
31   $ra     contains return address
```

- Some have usage conventions

```
         1     $at       reserved for pseudo-instructions
       2-3     $v0-$v1   return values of a function call
       4-7     $a0-$a3   arguments for a function call
8-15,24,25     $t0-$t9   temporaries, can be overwritten by function
     16-23     $s0-$s7   saved, have to be preserved by function
     26-27     $k0-$k1   reserved for kernel
        28     $gp       global area pointer
        29     $sp       stack pointer
        30     $fp       frame pointer
```
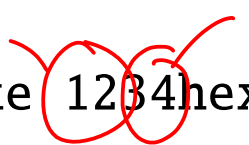
— access to heap

points to current stack frame

Endianness

- How are 16 bit numbers like 1234hex stored in memory?

Endianness

MSB *LSB*

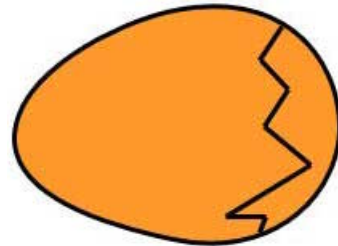- How are 16 bit numbers like 1234hex stored in memory?

Address	Little Endian	Big Endian
0000	34	12
0001	12	34

Endianness

- How are 16 bit numbers like 1234hex stored in memory?

Address	Little Endian	Big Endian
0000	34	12
0001	12	34

- From Jonathan Swift's "Gulliver's Travels" (1726):
  War over how to crack an egg:

Big Endian
People's tradition
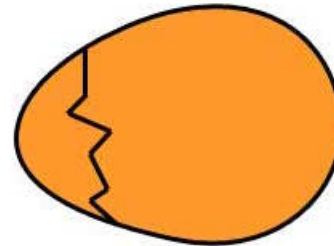
Little Endian
King's order

Endianness

- How are 16 bit numbers like 1234hex stored in memory?

Address	Little Endian	Big Endian
0000	34	12
0001	12	34

- From Jonathan Swift's "Gulliver's Travels" (1726):
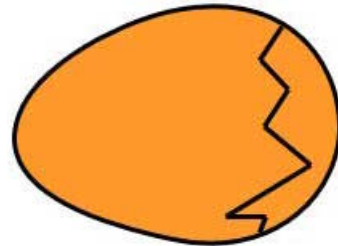 War over how to crack an egg:

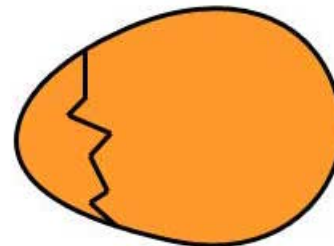Big Endian
People's tradition

Little Endian
King's order

- Little Endian: 6502, x86

- Big Endian: MIPS, Internet transfer protocols

instruction formats

- All instructions are encoded in 4 bytes --- 32 bits

- Instruction format (register type)

  - 6 bits: op: operation code
  - 5 bits: rs: first source operand register
  - 5 bits: rt: second source operand register
  - 5 bits: rd: return operand register
  - 5 bits: shamt: shift amount (for shift instructions)
  - 6 bits: funct: function code, indicates variant of operation

$$2^5 = 32$$

- Examples

  - add: operation code 0, function code 32
  - sub: operation code 0, function code 34

6502 : lda #$10

Instruction Format (I Type)

- Some operations may directly use 16 bit values

- Example: addi $s1, $s2, 100
  (adds value of register $s2 and 100, stores result in register $s1)

- Instruction format (immediate type)

  - 6 bits: op: operation code
  - 5 bits: rd: return operand register
  - 5 bits: rs: source operand register
  - 16 bits: constant or address

  not 32!

32 Bit Values

- All instructions are encoded in 32 bits

- Registers can hold 32 bit values

- How can we load 32 bit values into a register?

32 Bit Values

- All instructions are encoded in 32 bits

- Registers can hold 32 bit values

- How can we load 32 bit values into a register?

$\Rightarrow$ Solution: 2 instructions

32 Bit Values

- All instructions are encoded in 32 bits

- Registers can hold 32 bit values

- How can we load 32 bit values into a register?

$\Rightarrow$ Solution: 2 instructions

- First load upper order 16 bits (**l**oad **u**pper **i**mmediate)

$$\text{lui \$s0, 0061h}$$

32 Bit Values

- All instructions are encoded in 32 bits

- Registers can hold 32 bit values

- How can we load 32 bit values into a register?

$\Rightarrow$ Solution: 2 instructions

- First load upper order 16 bits (**l**oad **u**pper **i**mmediate)

$$\text{lui \$s0, 0061h}$$

- Then combine with lower order 16 bits (**or i**mmediate)

$$\text{ori \$s0, \$s0, 2304h}$$

- Stored value: 00612304h

- Jump instruction uses J Type format

  – 6 bits: operation code

  – 26 bits: address (relative) *effectively × 4*

- 26 bits, 4 byte increments → 256 MB address space

- There is also a "jump register" instruction

instructions

Computer Systems Fundamentals: MIPS Introduction

- Arithmetic: add, sub, mult, div

  HW

- Memory access: lb, sb

- Logic: and, or, not, xor

- Comparison: slt

- Branch: beq, bne

- Jumps: j, jal

- Instructions operate on varying data types

- 8 bits = 1 byte

- 16 bits = 2 bytes = 1 half word

- 32 bits = 4 bytes = 1 word

- 64 bits = 8 bytes = 2 words = 1 double word

Arithmetic

- Load immediately one number (s0 = 2)

  li $s0, 2

  pseudo instruction! ⟹ *ori*

- Add 4 (s1 = s0 + 4)

  addi $s1, $s0, 4

  immediate

- Subtract 3 (s2 = s1 - 3)

  addi $s2, $s1, -3

  add $s3, $s2, $si

  regs

- So far, assign absolute value to register

```
li $s0, 2
```

- So far, assign absolute value to register

$$\text{li } \$s0, \ 2$$

- Load value from memory address stored in register

$$\text{lw } \$s0, \ 0(\$s1)$$

— lw = load word (4 bytes)
— $s1 contains memory address
— 0(...) = offset 0

pointer is $s1

Memory Access

- So far, assign absolute value to register

$$\text{li } \$s0, 2$$

- Load value from memory address stored in register

$$\text{lw } \$s0, 0(\$s1)$$

  – lw = load word (4 bytes)
  – $s1 contains memory address
  – 0(...) = offset 0

- Bigger offset example: lw $s0, 8($s1)

  – word takes 4 bytes
  – offset 8

  5($s1)   useful?

  → 32 memory positions added

  no

- Cannot specify address directly

  - address takes 32 bits
  - instruction size is 32 bits

Direct Memory Access?

- Cannot specify address directly

  – address takes 32 bits
  – instruction size is 32 bits

- Workaround: store address in register first

- Cannot specify address directly

  – address takes 32 bits
  – instruction size is 32 bits

- Workaround: store address in register first

- 2 instructions needed:

```
lui $s1, 3264h
ori $s1, $s1, 8278h
```

  – address: 32648278h
  – first load upper memory address halfword (lui)
  – combine with lower memory address halfword (ori)

Direct Memory Access?

- Cannot specify address directly

  - address takes 32 bits
  - instruction size is 32 bits

- Workaround: store address in register first

- 2 instructions needed:

$$\text{lui \$s1, 3264h}$$
$$\text{ori \$s1, \$s1, 8278h}$$

  - address: 32648278h
  - first load upper memory address halfword (lui)
  - combine with lower memory address halfword (ori)

- Now retrieve value from that memory address

$$\text{lw \$s0, 0(\$s1)}$$

- We already encountered Boolean OR:

$$\texttt{ori \$s1, \$s1, 8278h}$$

- Register only version (s1 = s2 OR s3)

$$\texttt{or \$s1, \$s2, \$s3}$$

- Note: bitwise operation

$$\texttt{01010101 OR 11110000} \rightarrow \texttt{11110101}$$

Other Boolean Operators

- AND

$$\text{and \$s1, \$s2, \$s3}$$

- NOT

$$\text{not \$s1, \$s2}$$

- NOR

$$\text{nor \$s1, \$s2, \$s3}$$

- XOR

$$\text{xor \$s1, \$s2, \$s3}$$

- Shift left logical

$$\text{sll \$s1, \$s2, 4}$$

count

  – shifts all bits left by 4 positions

  – 0000 1001 $\rightarrow$ 1001 0000

  – equivalent to multiplication with 2^4

- Corresponding command: shift right logical (srl)

- No flags!

- Branch includes test

- Example

$$\text{beq } \$s1, \$s2, \text{ address}$$

  – beq = branch if equal
  – branches if registers $s1 and $s2 have same value

- Corresponding command: branch if not equal (bne)

Testing Inequality

- Another useful test: $s0 < $s1 ?

- Instruction: set on less than

$$\text{slt } \$s2, \$s0, \$s1$$

- Result: $s0 < $s1 $\rightarrow$ $s2 = 1 (otherwise 0)

- Can be used in branching

```
slt $s2, $s0, $s1
bne $s2, $zero, address
```

test result

Addressing in Branches

- Comparison of register values

```
beq register1, register2, address
```

Addressing in Branches

- Comparison of register values

$$\text{beq register1, register2, address}$$

- Format: I Type $\rightarrow$ address has 16 bits

- Address relative to current program counter

- Branches are typically local: 16 bits typically enough
  (also in 6502: 1 byte relative addressing)

spim

- Available at http://spimsimulator.sourceforge.net/

  – versions for Windows, Linux, Mac, etc.

- Installed on CS machines

- We will use this for homeworks

Basic Usage

- Write assembly program as text file

- Start the spim simulator

  ```
  % spim
  SPIM Version 7.3. of August 28, 2006
  Copyright 1990-2004 by James R. Larus (larus@cs.wisc.edu).
  (spim)
  ```

- Load program and step through the program

- Useful instructions:

  - load "countdown.s"
  - step
  - print $s0
  - reinitialize

Example Program

- Text file "countdown.s"

```
        .text

main:
    li $s0, 10              # store 10 in register $s0

  loop:
    addi $s0, $s0, -1       # decrement counter
    bne  $s0, $zero,loop    # != 0 ? then loop

exit:
    jr   $ra                # return to callee
```

- Loops through numbers 10 ... 0 in register $s0

Step through Example

- Ignore initial header code:

```
(spim) step
[0x00400000] 0x8fa40000  lw $4, 0($29)                    ; 175: lw $a0 0($sp) # argc
(spim)
[0x00400004] 0x27a50004  addiu $5, $29, 4                 ; 176: addiu $a1 $sp 4 # argv
(spim)
[0x00400008] 0x24a60004  addiu $6, $5, 4                  ; 177: addiu $a2 $a1 4 # envp
(spim)
[0x0040000c] 0x00041080  sll $2, $4, 2                    ; 178: sll $v0 $a0 2
(spim)
[0x00400010] 0x00c23021  addu $6, $6, $2                  ; 179: addu $a2 $a2 $v0
(spim)
[0x00400014] 0x0c100009  jal 0x00400024 [main]            ; 180: jal main
```

- This handles parameters from the command line

- First instruction

```
(spim) step
[0x00400024] 0x3410000a  ori $16, $0, 10    ; 4: li $s0, 10    # store 10 in register $s0
```

Step through Example

- First instruction

  ```
  (spim) step
  [0x00400024] 0x3410000a  ori $16, $0, 10     ; 4: li $s0, 10    # store 10 in register $s0
  ```

- Inspect value of register $s0

  ```
  (spim) print $s0
  Reg 16 = 0x0000000a (10)
  ```

Step through Example

- First instruction

  ```
  (spim) step
  [0x00400024] 0x3410000a  ori $16, $0, 10     ; 4: li $s0, 10    # store 10 in register $s0
  ```

- Inspect value of register $s0

  ```
  (spim) print $s0
  Reg 16 = 0x0000000a (10)
  ```

- Decrease loop index variable

  ```
  (spim) step
  [0x00400028] 0x2210ffff  addi $16, $16, -1   ; 7: addi $s0, $s0, -1     # decrement counter
  (spim) print $s0
  Reg 16 = 0x00000009 (9)
  ```

Step through Example

- Check loop termination condition

```
(spim) step
[0x0040002c] 0x1600ffff  bne $16, $0, -4 [loop-0x0040002c]; 8: bne  $s0, $zero,loop  # != 0 ? then loop
```

Step through Example

- Check loop termination condition

```
(spim) step
[0x0040002c] 0x1600ffff  bne $16, $0, -4 [loop-0x0040002c]; 8: bne  $s0, $zero,loop  # != 0 ? then loop
```

- Next iteration

```
(spim) step
[0x00400028] 0x2210ffff  addi $16, $16, -1  ; 7: addi $s0, $s0, -1     # decrement counter
(spim) print $s0
Reg 16 = 0x00000008 (8)

[...]

(spim)
[0x00400028] 0x2210ffff  addi $16, $16, -1  ; 7: addi $s0, $s0, -1     # decrement counter
(spim)
[0x0040002c] 0x1600ffff  bne $16, $0, -4 [loop-0x0040002c]; 8: bne  $s0, $zero,loop  # != 0 ? then loop
(spim)
[0x00400030] 0x03e00008  jr $31              ; 11: jr   $ra               # return to callee
```

- Termination

Print on Screen

- Print value of register $s0

- Place in loop:

```
move $a0, $s0              # value to print in $a0
li   $v0, 1                # print int
syscall
```

- Run in spim

```
(spim) reinitialize
(spim) load "countdown-and-print.s"
(spim) run
9876543210
```