CSCI 210: Computer Architecture
Lecture 5: Assembly continued

Stephen Checkoway
Oberlin College
Slides from Cynthia Taylor

CS History: Nintendo 64

- Released in 1996
- Named after its 64-bit CPU
- Silicon Graphics Inc (SGI) adapted MIPS chips for supercomputers to use less power and be cheaper
- "At the heart of the system will be a version of the MIPS Multimedia Engine, a chip-set consisting of a 64-bit MIPS RISC microprocessor, a graphics co-processor chip and Application Specific Integrated Circuits" (SGI press release, 1993)
- "If it works at all, it could bring MIPS to levels of volume [SGI] never dreamed of." (Michael Slater, Microprocessor Report)
- Super Mario 64 features a rabbit named MIPS after the processor

Review: Memory Instructions

- `ld  t0, 0(t1)`
  - `t0 = Mem[t1+0]`
  - Loads 8-byte double word from memory starting at address `t1+0` into register `t0`
- `sd  t0, 8(t1)`
  - `Mem[t1+8] = t0`
  - Stores 8-bytes double word in register `t0` into memory starting at address `t1+8`
- These instructions are the cornerstones of our being able to move data to and from memory

Review: If an 8-byte doubleword is in memory at address 4203088, what is the address of the next doubleword in memory?

A) 4203089
B) 4203096
C) 14203084
D) It depends on the value of the doublewords in memory
E) Since a doubleword is 8 bytes, it’s not possible to have one at address 4203088

Arrays

- Arrays are stored consecutively in memory
- The **base** address points to the first element in the array
- Accessing other elements in the array requires adding an **offset** to the base address
  - The offset to use is the index of the array element * the size of one element

Memory Operand Example 1

- C code:

```
g = h + A[6];
```

  – g in s1, h in s2, base address of A in s3, A is an array of 8 byte dwords

- RISC-V code:

  – Index 6 requires offset of 6 * 8 = 48

```
ld  t0, 48(s3)
add s1, s2, t0
```

- If the array contained 4-byte words instead, what would the offset be? What instruction would we use instead of `ld`?

Translate to RISC-V

- C code: g = h + A[5];
  - g in s1, h in s2, base address of A in s3.
  - A is an array of 4-byte words

A.
```
lw  t0, 5(s3)
add s1, s2, t0
```

B.
```
lw  t0, 20(s3)
add s1, s2, t0
```

C.
```
ld  t0, 20(s3)
add s1, s2, t0
```

D.
```
ld  t0, s3+5
add s1, s2, t0
```

Memory Operand Example 2

- C code:

```
A[12] = h + A[8];
```

  - h in s2, base address of A in s3, A is an array of 8-byte doublewords

- RISC-V code:

  - Index 8 requires offset of 8*8 = 64, index 12 requires an offset of 96

```
ld  t0, 64(s3)      # load doubleword
add t0, s2, t0
sd  t0, 96(s3)      # store doubleword
```

When a 2-byte integer is stored in byte-addressed memory (occupying two consecutive bytes), the most significant byte (MSB) is stored at one of the two consecutive locations in memory and the least significant byte (LSB) is stored in the other.

For example, 31337 as a 16-bit (2-byte) integer has the binary value 01111010_01101001 where 01111010 is the MSB and 01101001 is the LSB.

If 31337 is stored in memory at addresses 1234 and 1235, which byte of memory holds the MSB and which byte holds the LSB?

A. MSB is stored at address 1234; LSB at address 1235
B. LSB is stored at address 1234; MSB at address 1235
C. It depends [on what?]

Byte ordering

- Big-endian: Most significant byte in lowest address
  - MIPS, Network byte order, Motorola 68000, PowerPC (usually), …
- Little-endian: Most significant byte in highest address
  - **RISC-V** (usually), Intel x86, x86-64, ARM (usually), …
- Bi-endian: Both big and little endian supported (sometimes switchable at runtime!)
  - RISC-V, ARM, PowerPC, Alpha, SPARC, …
- Middle-endian/mixed-endian
  - Bytes not stored in either order, at least in some cases
  - Words stored one endian, bytes within words stored another endian
  - PDP-11 stored 16-bit words in little-endian order, but 32-bit double words as pairs of 16-bit words with the most significant word of the double word stored first (i.e., big-endian order)

Big-endian means most significant byte/digit/piece comes first, little-endian means least significant byte/digit/piece comes first. Mixed-endian means not in order.

Which row of the table correctly identifies the endianness of date formats?

	US (MM-DD-YYYY)	Most of the world (DD-MM-YYYY)	ISO 8601 (YYYY-MM-DD)
A	Little	Mixed	Big
B	Big	Little	Mixed
C	Mixed	Little	Big
D	Mixed	Big	Little
E	Little	Big	Mixed

Discussion question: Determine endianness

- We can store bytes, halfwords, words, and doublewords in memory using sb, sh, sw, and sd
- We can load bytes, halfwords, words, and doublewords from memory using lb, lh, lw, and ld

- In groups, discuss how you might use one of the store instructions and one of the load instructions to determine whether the particular RISC-V implementation is big-endian or little-endian
- Select any answer with your clicker when you have a solution

Questions about Memory?

Immediate Operands

- Constant data specified directly in an instruction
  - `addi  s3, s2, 4     # s3 = s2 + 4`
  - `li    t0, -25       # Pseudoinstruction: addi t0, zero, -25`
  - `ori   s0, t6, 1     # s0 = t6 | 1 (bit-wise OR, coming soon)`

Arithmetic and logic instructions with immediate operands

- `addi  dest, src, imm  # dest = src + imm`
- `andi  dest, src, imm  # dest = src & imm (bit-wise AND)`
- `ori   dest, src, imm  # dest = src | imm (bit-wise OR)`
- `xori  dest, src, imm  # dest = src ^ imm (bit-wise XOR)`

- The immediate operand `imm` is limited to 12 bits (we'll see why soon enough)
- This means it supports values in the range -2048–2047
- Larger values need to be put in registers first and the non-immediate version of the instruction (`add`, `and`, `or`, `xor`) is then used

Subtract 2 from s0 and store in register s1

A. `addi s0, s1, -2`

B. `addi s1, s0, -2`

C. `sub s0, s1, 2`

D. `subi s1, s0, 2`

E. More than one of the above

Pseudoinstructions

- `mv dest, src  => add dest, zero, src`
- `li dest, imm  => addi dest, zero, imm`
- `la dest, label  => auipc dest, …`
  `addi dest, dest, …`

The move (`mv`) instruction copies one register to another
The load immediate (`li`) instruction sets a register to an immediate value
The load address (`la`) instruction puts the memory address of a label in a register and it uses two instructions, `auipc` and `addi`

RISC-V Design Principles

- Simplicity favors regularity
  - fixed size instructions
  - small number of instruction formats
- Smaller is faster
  - limited instruction set
  - limited number of registers in register file
- Make the common case fast
  - arithmetic operands from the register file (load-store machine)
  - **allow instructions to contain small immediate operands**

Loading large numbers into registers

- Immediate operands are limited to 12 bits: -2048 to 2047
- Numbers outside this range need to be loaded into registers before they can be used
- RISC-V has two special instructions, lui and auipc, which it uses for this purpose (in two different situations)

Load upper immediate (lui)

- The `lui dest, imm` instruction has a special format that supports a 20-bit immediate value that it loads into bits 31–12 of the destination register
- This is usually paired with the `addi` instruction to add the least significant 12 bits
- Together, these can load an arbitrary 32-bit word into a register
- The assembler will translate `li dest, imm` into a `lui`/`addi` pair if `imm` is too large
- Example: `li t0, 305419896` becomes

```
lui   t0, 74565
addi  t0, 1656
```

Add upper immediate to pc (auipc)

- The `la dest, label` pseudoinstruction sets the destination register to the address of the label
- A memory address is 64-bits so the same trick with lui/addi won't work
- auipc dest, offset instruction shifts the 20-bit offset left by 12 bits and adds that to the address of the instruction itself (the value of the pc register)
- The assembler expands la into an `auipc`/`addi` pair of instructions
- The `la a0, msg` from the hello world example becomes

```
auipc  a0, 64528
addi   a0, a0, -4
```

```
&msg = &auipc + (64528 << 12) - 4
```

RISC-V Questions?

Reading

- Next lecture: Number Representation
  - Section 2.4

- Problem Set 0 – due today
- Problem Set 1 – due next Friday (you can start now)

- Self-study
  - Assemble a program in RARS using the `la` pseudoinstruction, investigate the immediate values used in the `auipc` and `addi` instructions `la` expands to. Look at the binary values of those immediates. Compare those to the addresses of the auipc instruction and the label
  - Assemble a program in RARS using the `li` pseudoinstruction with differently sized immediate values, including those larger than 32-bits. Look at the generated code to see how the assembler loads those values into registers