Chapter 6 :: Architecture

Quiz 1 & 2

① → 6 $2^6 = 64$

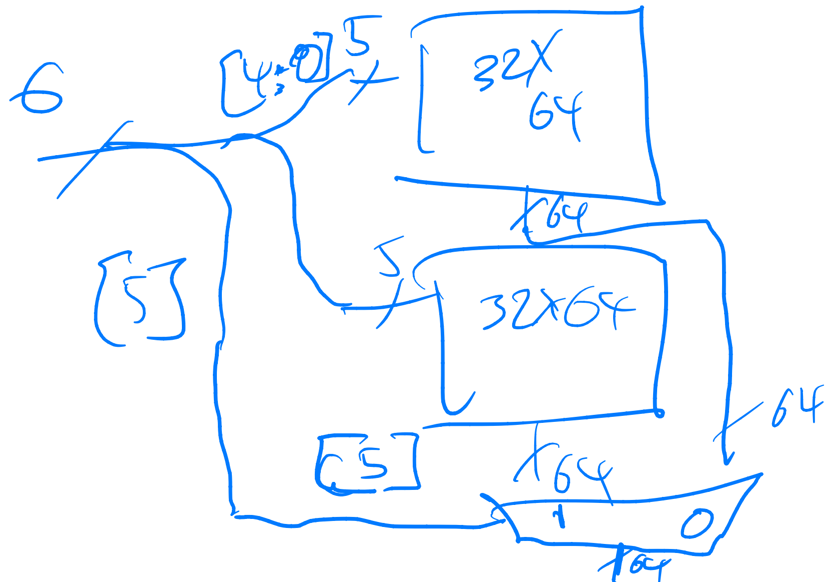
③ SRAM is faster
DRAM takes less area

6.1-6.3 MIPS instructions

Digital Design and Computer Architecture

David Money Harris and Sarah L. Harris

②



000
001
010
011
100
101
110
111

Introduction

- **Architecture:** the programmer's view of the computer
  - Defined by instructions (operations) and operand locations
- **Microarchitecture:** how to implement an architecture in hardware (covered in Chapter 7)

Application Software	programs
Operating Systems	device drivers
Architecture	instructions registers
Micro-architecture	datapaths controllers
Logic	adders memories
Digital Circuits	AND gates NOT gates
Analog Circuits	amplifiers filters
Devices	transistors diodes
Physics	electrons

Assembly Language

- To command a computer, you must understand its language.
  - **Instructions:** words in a computer's language → smallest unit of computation
  - **Instruction set:** the vocabulary of a computer's language ISA
- Instructions indicate the operation to perform and the operands to use.
  - **Assembly language:** human-readable format of instructions
  - **Machine language:** computer-readable format (1's and 0's)
- MIPS architecture:
  - Developed by John Hennessy and his colleagues at Stanford and in the 1980's.
  - Used in many commercial systems, including Silicon Graphics, Nintendo, and Cisco
- Once you've learned one architecture, it's easy to learn others.

Instructions: Addition

High-level code

```
a = b + c;
```

MIPS assembly code

```
add a, b, c
```

- add: mnemonic indicates what operation to perform
- b, c: source operands on which the operation is performed
- a: destination operand to which the result is written

Instructions: Subtraction

- Subtraction is similar to addition. Only the mnemonic changes.

High-level code

```
a = b - c;
```

MIPS assembly code

```
sub a, b, c
```

- sub: mnemonic indicates what operation to perform
- b, c: source operands on which the operation is performed
- a: destination operand to which the result is written

Multiple Instructions

- More complex code is handled by multiple MIPS instructions.

t= temporary

C Code

```
a = b + c - d;
```

MIPS assembly code

```
add t, b, c  # t = b + c
sub a, t, d  # a = t - d
```

Operands

- Operand location: physical location in computer
  - Registers "working memory"
  - Memory
  - Constants (also called *immediates*)

Operands: Registers

- MIPS has 32 32-bit registers
- Registers are faster than memory
- MIPS called "32-bit architecture" because it operates on 32-bit data

Operands: Registers

- Registers:
  - Written with a dollar sign (`$`) before their name
  - For example, register 0 is written "`$0`", pronounced "register zero" or "dollar zero".
- Certain registers used for specific purposes:
  - For example,
    - `$0` always holds the constant value 0.
    - the *saved registers*, `$s0–$s7`, are used to hold variables
    - the *temporary registers*, `$t0 - $t9`, are used to hold intermediate values during a larger computation.
- For now, we only use the temporary registers (`$t0 - $t9`) and the saved registers (`$s0 - $s7`).
- We will use the other registers in later slides.

MIPS Register Set

register address

Name	~~Register Number~~	Usage
`$0`	0	the constant value 0
`$at`	1	assembler temporary
`$v0-$v1`	2-3	Function return values
`$a0-$a3`	4-7	Function arguments
`$t0-$t7`	8-15 01000->01111	temporaries
`$s0-$s7`	16-23 10000->10111	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	Function return address

MIPS Registers

Register name	Number	Usage
zero	0	constant 0
at	1	reserved for assembler
v0, v1	2 ~ 3	expression evaluation and results of a function
a0 ~ a3	4 ~ 7	arguments 1 - 4
t0 ~ t7	8 ~ 15	temporary (not preserved across call)
s0 ~ s7	16 ~ 23	saved (preserved across call)
t8, t9	24, 25	temporary (not preserved across call)
k0, k1	26, 27	reserved for OS kernel
gp	28	pointer for global area
sp	29	stack pointer
fp	30	frame pointer
ra	31	return address (used by function call)

Stop here 9/8

Instructions with registers

- Revisit add instruction

s
t

High-level code

```
a = b + c
```

MIPS assembly code

```
# $s0 = a, $s1 = b, $s2 = c
add $s0, $s1, $s2
```

destination

source

actual MIPS

Instructions with registers

- Revisit complex instruction

High-level code

```
a = b + c - d
```

MIPS assembly code

```
# $s0 = a, $s1 = b, $s2 = c,
   $s3 = d $t0 = t
add $t0, $s1, $s2
sub $s0, $t0, $s3
```

Operands: Memory

- Too much data to fit in only 32 registers
- Store more data in memory
- Memory is large, so it can hold a lot of data

larger = slower

- But it's also slow
- Commonly used variables kept in registers → "working data"
- Using a combination of registers and memory, a program can access a large amount of data fairly quickly

Word-Addressable Memory

- Each 32-bit data word has a unique address

like our typical memory arrays

Word Address	Data	
⋮	⋮	⋮
0x 00000003	4 0 F 3 0 7 8 8	Word 3
0x 00000002	0 1 E E 2 8 4 2	Word 2
0x 00000001	F 2 F 1 A C 0 7	Word 1
0x 00000000	A B C D E F 7 8	Word 0

Note: MIPS uses byte-addressable memory, which we'll talk about next.

Byte-Addressable Memory

- Each data byte has a unique address
- Load/store words or single bytes: load byte (`lb`) and store byte (`sb`)
- **Each 32-bit words has 4 bytes, so the word address increments by 4**

1C
18
14
10

Word Address	Data				
⋮	⋮				⋮
0000000C	4 0	F 3	0 7	8 8	Word 3
00000008	0 1	E E	2 8	4 2	Word 2
00000004	F 2	F 1	A C	0 7	Word 1
00000000	A B	C D	E F	7 8	Word 0

MSB width = 4 bytes LSB

0 1 2 3 ← big endian
3 2 1 0 ← little endian

MIPS is big-endian

Reading Byte-Addressable Memory

- The address of a memory word must now be multiplied by 4. For example,
  - the address of memory word 2 is 2 × 4 = 8
  - the address of memory word 10 is 10 × 4 = 40 (0x28)
- Load a word of data at memory address 4 into `$s3`.
- `$s3` holds the value 0xF2F1AC07 after the instruction completes.
- **MIPS is byte-addressed, not word-addressed**

MIPS assembly code

```
lw $s3, 4($0)   # read word at address 4 into $s3
```

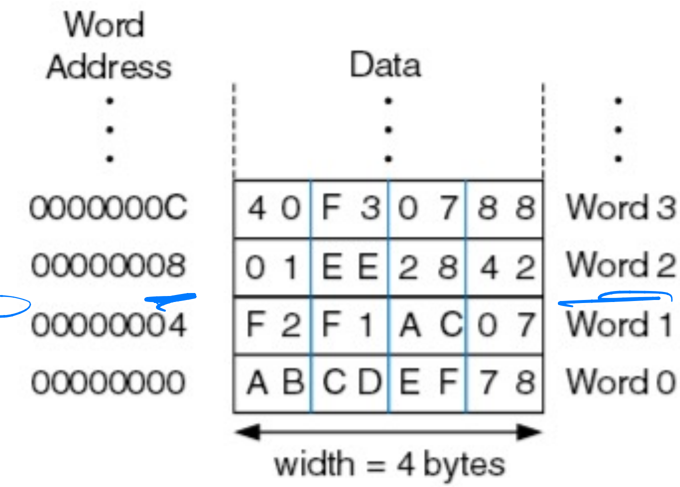
4 + 0

destination

Word Address	Data	
⋮	⋮	⋮
0000000C	4 0 F 3 0 7 8 8	Word 3
00000008	0 1 E E 2 8 4 2	Word 2
00000004	F 2 F 1 A C 0 7	Word 1
00000000	A B C D E F 7 8	Word 0

width = 4 bytes

Each 32-bit words has 4 bytes, so the word address increments by 4



destination

source: offset(base address)

source = base+offset

load word: load from memory

store word: store into memory

Code Example 6.8 ACCESSING BYTE-ADDRESSABLE MEMORY

MIPS Assembly Code

```
lw $s0, 0($0)        # read data word 0 (0xABCDEF78) into $s0
lw $s1, 8($0)        # read data word 2 (0x01EE2842) into $s1
lw $s2, 0xC($0)      # read data word 3 (0x40F30788) into $s2
sw $s3, 4($0)        # write $s3 to data word 1
sw $s4, 0x20($0)     # write $s4 to data word 8
sw $s5, 400($0)      # write $s5 to data word 100
```

source

destination=base+offset

Operands: Constants/Immediates

- `lw` and `sw` illustrate the use of constants or *immediates*
- Called immediates because they are *immediate*ly available from the instruction
- Immediates don't require a register or memory access.
- The add immediate (`addi`) instruction adds an immediate to a variable (held in a register).
- An immediate is a 16-bit two's complement number.
- Is subtract immediate (`subi`) necessary?

High-level code

```
a = a + 4;
b = a - 12;
```

MIPS assembly code

```
# $s0 = a, $s1 = b
addi $s0, $s0, 4
addi $s1, $s0, -12
```