ECE 550D
Fundamentals of Computer Systems and Engineering

Fall 2023

Instruction Set Architectures (ISAs) and MIPS
(Microprocessor without Interlocked Pipeline Stages)

Xin Li & Dawei Liu
Duke Kunshan University

Slides are derived from work by
Andrew Hilton, Tyler Bletsch and Rabih Younes (Duke)

Last time…

- Who can remind us what we did last time?
  - Finite State Machines
  - Division

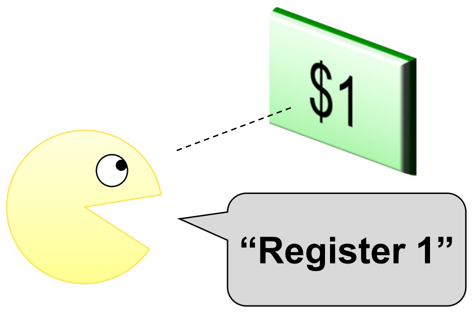
Now: Moving to software

- C is pretty low level
  - Most of you: in 551, can program in C
    - Others: assume you know programming, pointers, etc.
  - But compiled into **assembly**…
- Lower level programming
  - What does assembly look like?
  - How does software communicate with hardware?
  - What variations can there be?
  - Advantages/disadvantages?

Assembly

- Assembly programming:
  - 1 **machine instruction** at a time
  - Still in "human readable form"
    - `add $1, $2, $3`
  - Much "lower level" than any other programming
    - Limited number of **registers** vs unlimited variables
    - Flat scope

How to say "$1" out loud:



Registers

- Two places processors can store data
  - Registers (saw these---sort of):
    - In processor
    - Few of them (e.g., 32)
    - Fast (more on this much later in semester)
  - Memory (later):
    - Outside of processor
    - Huge (e.g., 16GB)
    - Slow (generally about 100—200x slower than registers, more later)
- For now: think of registers like “variables”
  - But only 32 of them
  - E.g., $1 = $2 + $3   much like x = y + z

Simple, Running Example

```
// silly C code

int sum, temp, x, y;
while (true){
    temp = x + y;
    sum = sum + temp;
}
```

```
// equivalent MIPS assembly code

loop:   lw $1, Memory[1004]
        lw $2, Memory[1008]
        add $3, $1, $2
        add $4, $4, $3
        j loop
```

Memory references don’t quite work like this…we’ll correct this later.

OK, so what does this assembly code mean?
Let’s dig into each line …

Simple, Running Example

```
loop:   lw $1, Memory[1004]
        lw $2, Memory[1008]
        add $3, $1, $2
        add $4, $4, $3
        j loop
```

NOTES
"loop:" = line label (in case we need to refer to this instruction's PC)
lw = "load word" = read a word (32 bits) from memory
$1 = "register 1" → put result read from memory into register 1
Memory[1004] = address in memory to read from (where x lives)

Note: PC means "program counter"
Note: almost all MIPS instructions put destination (where result gets written) first (in this case, $1)

Simple, Running Example

```
loop:   lw $1, Memory[1004]
        lw $2, Memory[1008]
        add $3, $1, $2
        add $4, $4, $3
        j loop
```

NOTES
lw = "load word" = read a word (32 bits) from memory
$2 = "register 2" → put result read from memory into register 2
Memory[1008] = address in memory to read from (where y lives)

Simple, Running Example

```
loop:   lw $1, Memory[1004]
        lw $2, Memory[1008]
        add $3, $1, $2
        add $4, $4, $3
        j loop
```

NOTES
add $3, $1, $2= add what's in $1 to what's in $2 and put result in $3

Simple, Running Example

```
loop:   lw $1, Memory[1004]
        lw $2, Memory[1008]
        add $3, $1, $2
        add $4, $4, $3
        j loop
```

NOTES
add $4, $4, $3= add what's in $4 to what's in $3 and put result in $4

Note: this instruction overwrites previous value in $4

Simple, Running Example

```
loop:   lw $1, Memory[1004]
        lw $2, Memory[1008]
        add $3, $1, $2
        add $4, $4, $3
        j loop
```

NOTES
j = "jump"
loop = PC of instruction at label "loop" (the first lw instruction above)
sets next PC to the address labeled by "loop"

Note: all other instructions in this code set next PC = PC++

Assembly too high level for machine

- Human readable is not (easily) machine executable
  - `add $1, $2, $3`
- **Instructions are numbers too!**
  - Bit fields (like FP numbers)
- Instruction Format
  - Establishes a mapping from "instruction" to binary values
  - Which bit positions correspond to which parts of the instruction (operation, operands, etc.)
- In MIPS, each assembly instruction has a unique 32-bit representation:
  - `add $3, $2, $7     ←→ 00000000010001110001100000100000`
  - `lw $8, Mem[1004]   ←→ 10001100000010000000001111101100`
- **Assembler** does this translation
  - Humans don't typically need to write raw bits

What Must be Specified?

- Instruction "opcode"
  - What should this operation do? (add, subtract,…)
- Location of operands and result
  - Registers (which ones?)
  - Memory (what address?)
  - Immediates (what value?)
- Data type and Size
  - Usually included in opcode
  - E.g., signed vs unsigned int (if it matters)
- What instruction comes next?
  - Sequentially next instruction, or jump elsewhere

The ISA

- Instruction Set Architecture (ISA)
  - **Contract between hardware and software**
  - Specifies everything hardware and software need to agree on
    - Instruction encoding and effects
    - Memory structure
    - Etc.

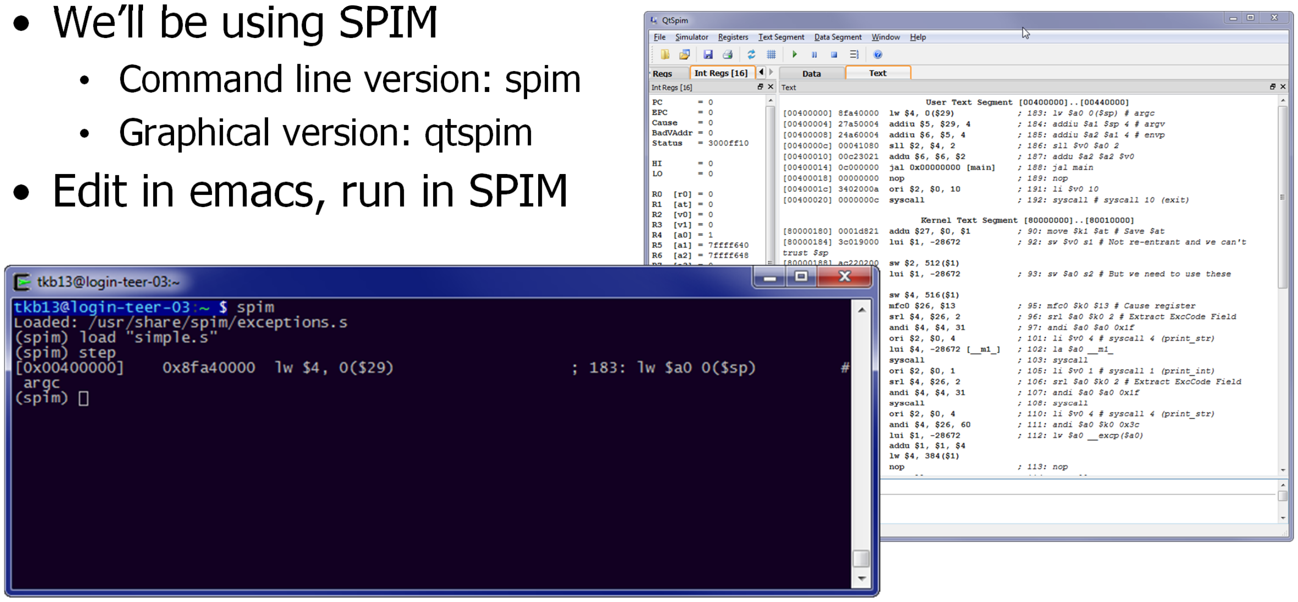
- Many different ISAs
  - x86 and x86_64 (Intel and AMD)
  - POWER (IBM)
  - **MIPS**
  - ARM
  - SPARC (Oracle)

Our focus: MIPS

- We will work with MIPS
  - x86 is ugly (x86_64 is less ugly, but still nasty)
  - MIPS is relatively "clean"
    - More on this in a minute

But I don't have a MIPS computer?

- We'll be using SPIM
  - Command line version: spim
  - Graphical version: qtspim
- Edit in emacs, run in SPIM


QtSpim
File Simulator Registers Text Segment Data Segment Window Help
Regs
Int Regs [16]
Data
Text
tkb13@login-teer-03:~

```
PC       = 0
EPC      = 0
Cause    = 0
BadVAddr = 0
Status   = 3000ff10

HI       = 0
LO       = 0

R0  [r0] = 0
R1  [at] = 0
R2  [v0] = 0
R3  [v1] = 0
R4  [a0] = 1
R5  [a1] = 7ffff640
R6  [a2] = 7ffff648
```

```
                    User Text Segment [00400000]..[00440000]
[00400000] 8fa40000  lw $4, 0($29)           ; 183: lw $a0 0($sp) # argc
[00400004] 27a50004  addiu $5, $29, 4        ; 184: addiu $a1 $sp 4 # argv
[00400008] 24a60004  addiu $6, $5, 4         ; 185: addiu $a2 $a1 4 # envp
[0040000c] 00041080  sll $2, $4, 2           ; 186: sll $v0 $a0 2
[00400010] 00c23021  addu $6, $6, $2         ; 187: addu $a2 $a2 $v0
[00400014] 0c000000  jal 0x00000000 [main]   ; 188: jal main
[00400018] 00000000  nop                     ; 189: nop
[0040001c] 3402000a  ori $2, $0, 10          ; 191: li $v0 10
[00400020] 0000000c  syscall                 ; 192: syscall # syscall 10 (exit)

                    Kernel Text Segment [80000000]..[80010000]
[80000180] 0001d821  addu $27, $0, $1        ; 90: move $k1 $at # Save $at
[80000184] 3c019000  lui $1, -28672          ; 92: sw $v0 s1 # Not re-entrant and we can't
trust $sp
[80000188] ac220200  sw $2, 512($1)
                     lui $1, -28672          ; 93: sw $a0 s2 # But we need to use these

                     sw $4, 516($1)
                     mfc0 $26, $13           ; 95: mfc0 $k0 $13 # Cause register
                     srl $4, $26, 2          ; 96: srl $a0 $k0 2 # Extract ExcCode Field
                     andi $4, $4, 31         ; 97: andi $a0 $a0 0x1f
                     ori $2, $0, 4           ; 101: li $v0 4 # syscall 4 (print_str)
                     lui $4, -28672 [__m1_]  ; 102: la $a0 __m1_
                     syscall                 ; 103: syscall
                     ori $2, $0, 1           ; 105: li $v0 1 # syscall 1 (print_int)
                     srl $4, $26, 2          ; 106: srl $a0 $k0 2 # Extract ExcCode Field
                     andi $4, $4, 31         ; 107: andi $a0 $a0 0x1f
                     syscall                 ; 108: syscall
                     ori $2, $0, 4           ; 110: li $v0 4 # syscall 4 (print_str)
                     andi $4, $26, 60        ; 111: andi $a0 $k0 0x3c
                     lui $1, -28672          ; 112: lw $a0 __excp($a0)
                     addu $1, $1, $4
                     lw $4, 384($1)
                     nop                     ; 113: nop
```

```
tkb13@login-teer-03:~ $ spim
Loaded: /usr/share/spim/exceptions.s
(spim) load "simple.s"
(spim) step
[0x00400000]    0x8fa40000  lw $4, 0($29)                 ; 183: lw $a0 0($sp)        #
 argc
(spim) 
```

- Get QtSPIM at: http://spimsimulator.sourceforge.net/

ISAs: RISC vs CISC

- Two broad categories of ISAs:
  - Complex Instruction Set Computing
    - Came first, days when people always directly wrote assembly
    - Big complex instructions
  - Reduced Instruction Set Computing
    - Goal: make hardware simple and fast
    - Write in high level language, let compiler do the dirty work
      - Rely on compiler to optimize for you
- Note:
  - Sometimes fuzzy: ISAs may have some features of each
  - Common mis-conception: not about how many different insns!

ISAs: RISC vs CISC

Reduced **I**nstruction **S**et **C**omputing

- Simple, fixed length instruction encoding
- Few **memory addressing modes**
- "Many" registers (e.g., 32)
- Three-operand arithmetic (dest = src1 op src2)
- **Load-store** ISA

ISAs: RISC vs CISC

- **C**omplex **I**nstruction **S**et **C**omputing
  - Variable length instruction encoding (sometimes quite complex)
  - Many addressing modes, some quite complex
  - Few registers (e.g., 8)
  - Various operand models
    - Stack
    - Two-operand (dest = src op dest)
    - Implicit operands
  - Can operate directly on memory
    - Register = Memory op Register
    - Memory = Memory op Register
    - Memory = Memory op Memory

Memory Addressing Modes

- Memory location: how to specify address
  - Simple (RISC)
    - Register + Immediate (e.g., address = $4 + 16)
    - Register + Register (e.g., address= $4 + $7)
  - Complex (CISC)
    - Auto-increment (e.g., address = $4; $4 = $4 + 4;)
    - Scaled Index (e.g., address = $4 + ($2 <<2) + 0x1234)
    - Memory indirect (e.g., address = memory[$4])

Load-Store ISA

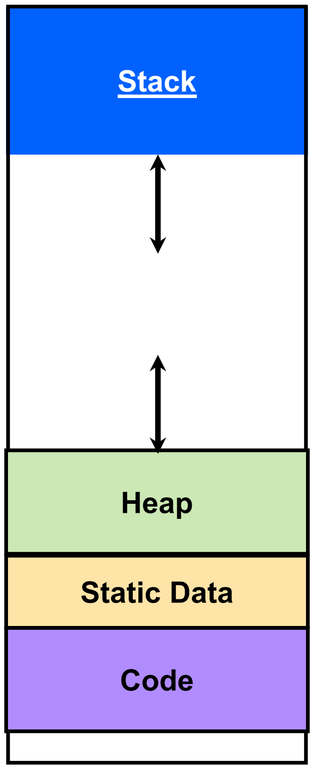
- Load-store ISA (RISC):
  - Specific instructions (loads/stores) to access memory
    - Loads read memory (and **only** read memory)
    - Stores write memory (and **only** write memory)
- Contrast with (CISC)
  - General memory operands ($4 = mem[$5] + $3)
  - Memory/memory operations: mem[$4] = mem[$5] + $3

Stored Program Computer

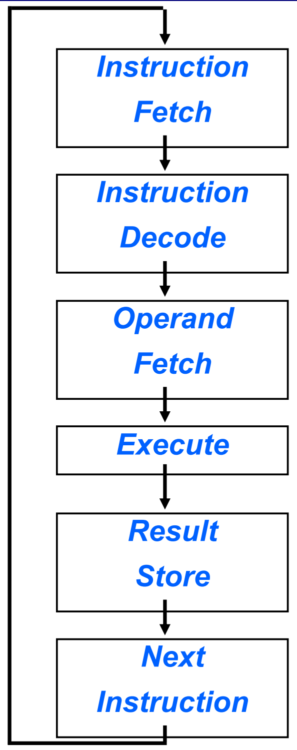
- Instructions: a fixed set of built-in operations
- Instructions and data are stored in memory
  - Allows general purpose computation!
- Fetch-Execute Cycle

```
while (!done)
    fetch instruction
    execute instruction
```

- Effectively what hardware does
- This is what the SPIM Simulator does



How are Instructions Executed?



- Instruction Fetch:
  Read instruction bits from memory
- Decode:
  Figure out what those bits mean
- Operand Fetch:
  Read registers (+ mem to get sources)
- Execute:
  Do the actual operation (e.g., add the #s)
- Result Store:
  Write result to register or memory
- Next Instruction:
  Figure out mem addr of next insn, repeat

More Details on Execution?

- Previous slide provides a high-level overview
  - Called von Neumann model
  - John von Neumann: Eniac (first digital computer)
- More details: How hardware works
  - Later in the course

- Now, diving into assembly programming/MIPS

Assembly Programming

- How do you write an assembly program?
- How do you write a program (in general)?

5 Step Plan (ECE 551)

- 5 Steps to write any program:
  1. Work an example yourself
  2. Write down what you did
  3. Generalize steps from 2
  4. Test generalized steps on different example
  5. Translate generalized steps to code

How to Write a Program

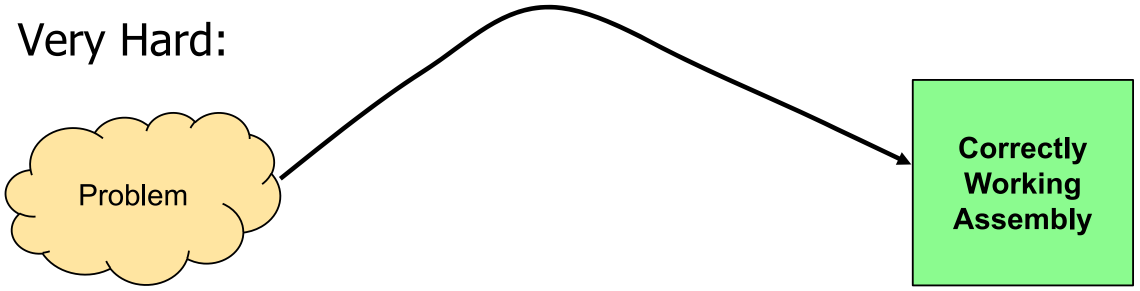
- How I teach programming:
  1. Work an example yourself
  2. Write down what you did
  3. Generalize steps from 2
  4. Test generalized steps on different example
  5. Translate generalized steps to code

Develop Algorithm In (Familiar) Higher Level Language

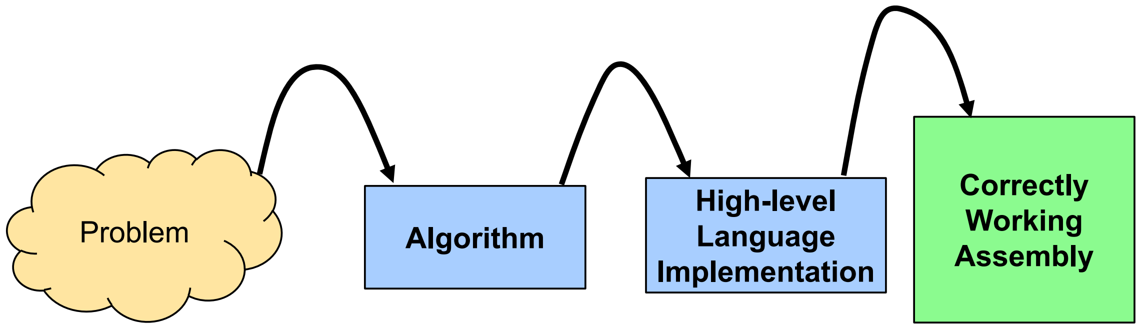
Then translate to lower level language

Why do I bring this up?

- Very Hard:

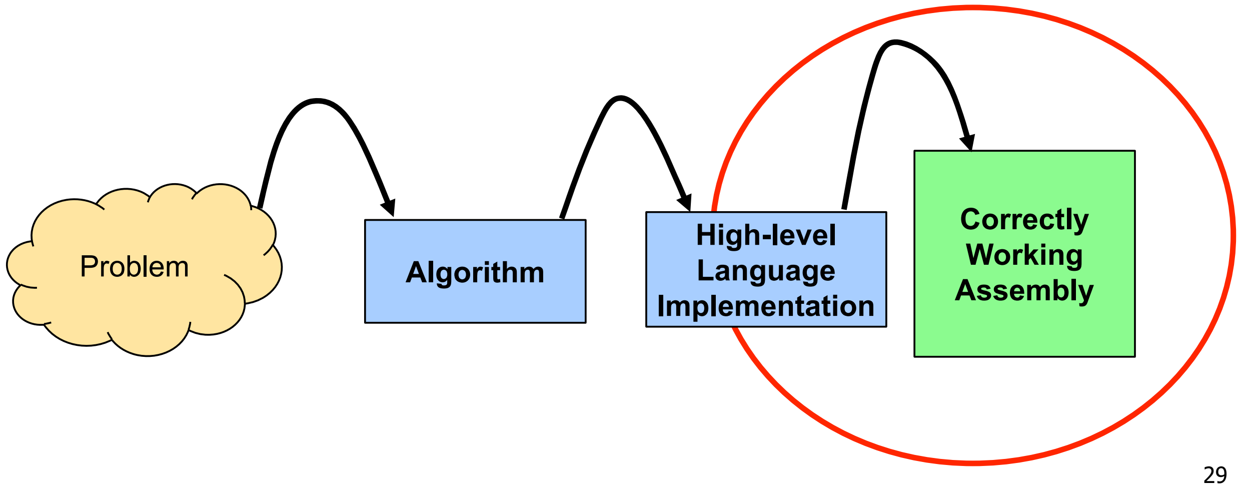


- Easier:


Problem
Algorithm
High-level Language Implementation
Correctly Working Assembly

Our focus

- We will focus on the assembly step
  - Assume you know how to devise an algorithm for a problem
    - I'll use C.


Problem
Algorithm
High-level Language Implementation
Correctly Working Assembly

Simplest Operations We Might Want?

- What is the simplest computation we might do?
  - Add two numbers:

x = a + b;

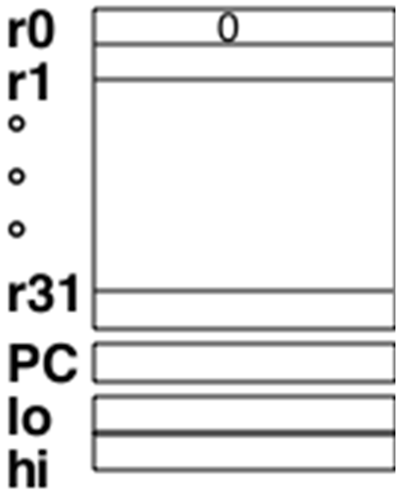
add $1, $2, $3

"Add $2 + $3, and store it in $1"

Note: when writing assembly, basically pick reg for a, reg for b, reg for x
Not enough regs for all variables? We'll talk about that later…

MIPS Integer Registers

- Recall: registers
  - Fast
  - In CPU
  - Directly compute on them
- 31 x 32-bit GPRs (R0 = 0)
  - GPRs: general purpose registers
- Also floating point registers
- A few special purpose regs too
  - PC = Address of next insn



Executing Add

Address	Instruction
1000	add $8, $4, $6
1004	add $5, $8, $7
1008	add $6, $6, $6
100C	…
1010	…

Register	Value
$0	0000 0000
$1	1234 5678
$2	C001 D00D
$3	1BAD F00D
$4	9999 9999
$5	ABAB ABAB
$6	0000 0001
$7	0000 0002
$8	0000 0000
$9	0000 0000
$10	0000 0000
$11	0000 0000
$12	0000 0000
…	…
PC	**0000 1000**

PC tells us where to execute next

Executing Add

Address	Instruction
1000	add $8, $**4**, $**6**
1004	add $5, $8, $7
1008	add $6, $6, $6
100C	…
1010	…

Add reads its source registers, and uses their values directly ("register direct")

9999 9999
<u>0000 0001</u>
9999 999A

Register	Value
$0	0000 0000
$1	1234 5678
$2	C001 D00D
$3	1BAD F00D
$4	**9999 9999**
$5	ABAB ABAB
$6	**0000 0001**
$7	0000 0002
$8	0000 0000
$9	0000 0000
$10	0000 0000
$11	0000 0000
$12	0000 0000
…	…
PC	0000 1000

Executing Add

Address	Instruction
1000	add $**8**, $4, $6
1004	add $5, $8, $7
1008	add $6, $6, $6
100C	…
1010	…

Add writes its result to its destination register

Register	Value
$0	0000 0000
$1	1234 5678
$2	C001 D00D
$3	1BAD F00D
$4	9999 9999
$5	ABAB ABAB
$6	0000 0001
$7	0000 0002
$8	**9999 999A**
$9	0000 0000
$10	0000 0000
$11	0000 0000
$12	0000 0000
…	…
PC	0000 1000

Executing Add

Address	Instruction
1000	add $8, $4, $6
1004	add $5, $8, $7
1008	add $6, $6, $6
100C	…
1010	…

And goes to the sequentially next instruction (PC = PC + 4)

Register	Value
$0	0000 0000
$1	1234 5678
$2	C001 D00D
$3	1BAD F00D
$4	9999 9999
$5	ABAB ABAB
$6	0000 0001
$7	0000 0002
$8	9999 999A
$9	0000 0000
$10	0000 0000
$11	0000 0000
$12	0000 0000
…	…
PC	**0000 1004**

Executing Add

Address	Instruction
1000	add $8, $4, $6
1004	add $5, $8, $7
1008	add $6, $6, $6
100C	…
1010	…

You all do the next instruction!

Register	Value
$0	0000 0000
$1	1234 5678
$2	C001 D00D
$3	1BAD F00D
$4	9999 9999
$5	ABAB ABAB
$6	0000 0001
$7	0000 0002
$8	9999 999A
$9	0000 0000
$10	0000 0000
$11	0000 0000
$12	0000 0000
…	…
PC	0000 1004

Executing Add

	Address	Instruction
	1000	add $8, $4, $6
	1004	add $5, $8, $7
➡	1008	add $6, $6, $6
	100C	…
	1010	…

Register	Value
$0	0000 0000
$1	1234 5678
$2	C001 D00D
$3	1BAD F00D
$4	9999 9999
$5	**9999 999C**
$6	0000 0001
$7	0000 0002
$8	9999 999A
$9	0000 0000
$10	0000 0000
$11	0000 0000
$12	0000 0000
…	…
PC	**0000 1008**

We set $5 equal to
$8 (9999 999A) + $7 (2) = 9999 999C

and PC = PC +4

Executing Add

	Address	Instruction
	1000	add $8, $4, $6
	1004	add $5, $8, $7
➡	1008	add $6, $6, $6
	100C	…
	1010	…

Register	Value
$0	0000 0000
$1	1234 5678
$2	C001 D00D
$3	1BAD F00D
$4	9999 9999
$5	9999 999C
$6	0000 0001
$7	0000 0002
$8	9999 999A
$9	0000 0000
$10	0000 0000
$11	0000 0000
$12	0000 0000
…	…
PC	0000 1008

Its perfectly fine to have $6 as a src and a dst
This is just like x = x + x; in C, Java, etc:

1 + 1 = 2

Executing Add

Address	Instruction
1000	add $8, $4, $6
1004	add $5, $8, $7
1008	add $6, $6, $6
100C	…
1010	…

Register	Value
$0	0000 0000
$1	1234 5678
$2	C001 D00D
$3	1BAD F00D
$4	9999 9999
$5	9999 999C
$6	**0000 0002**
$7	0000 0002
$8	9999 999A
$9	0000 0000
$10	0000 0000
$11	0000 0000
$12	0000 0000
…	…
PC	**0000 100C**

Its perfectly fine to have $6 as a src and a dst
This is just like x = x + x; in C, Java, etc:

1 + 1 = 2

Summary

- MIPS ISA and Assembly Programming
- Continue on next lecture…