Advanced Code: An Introduction to Assembly Code and Interpreters 5/8/19

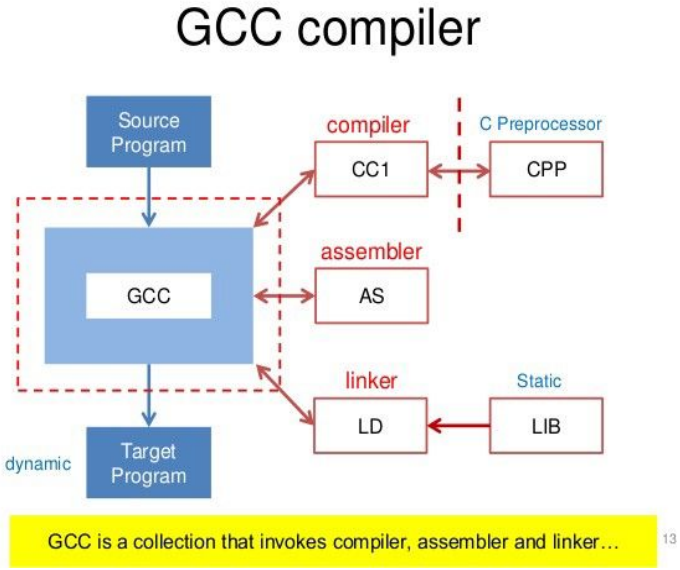
Ryan Toner

Learning Objective

1. Introduce you to the basics of Assembly — particularly through the lens of MIPS and the architecture of the MIPS processor.
2. Relate your understanding of Assembly to the higher level code you already know.
3. Teach you how to write basic MIPS code while refreshing your memory on binary data and the importance of abstraction.
4. Explain the concept of an interpreter and how it differs from compiled code.
    a. Demonstrate a working MIPS interpreter and simulator.
    b. Introduce you to the idea of a Pipelined architecture
5. Work on a 3 problem lab individually using assembly!
    a. We will stop at some point to go over solutions!

What is Assembly?

- We Write in High Level Languages
- These get *Compiled* into assembly relative to the manufacturer's architecture
- An *Assembler* takes assembly into machine code (only binary!)
- A *Linker* creates the executable, particularly by ensuring any used libraries make their way to the final program



Speaking to the Computer

- Instruction Set/ ISA -- commands to tell the processor what to do
- An abstract model of the computer, but still closely tied to the computer's logical design
- Interface between the software we write and the hardware that runs it
- We can write in assembly language, which is easily translated to the ISA of your computer

- **With great power comes great responsibility**

Representing Data in MIPS

- Byte = 8 bits
- Word = 4 bytes = 32 bits
- MIPS uses 32-bit architecture
- Registers are the most fundamental unit of storing data
  - Used by processor for computing
  - Stores 32 bits
- Addressing: delineate the location of data in memory

*In assembly, can we perform arithmetic operations on **addressed** data?*

Data Transfer Instructions (like loading) must be used for us to use the processor to manipulate our data!

Refresher: Relationship between bits and data

- Binary data, on its own, is meaningless.
- We give it meaning by, at the higher levels, telling the computer how to understand that data. As programmer's, we use String, Int, Long, Float, etc regularly to tell the computer how to use that data.
- As such, our registers in MIPS can truly store anything in a similar regard! But be careful, because you should remember what data type you are using in a register (especially when printing!)

MIPS operands

Name	Example	Comments
32 registers	`$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic, register `$zero` always equals 0, and register `$at` is reserved by the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers.

Useful Registers

- Save Registers
- Temporary Registers
- $zero
- $sp
- $ra
- Argument Registers
- Value Registers

Register Number	Mnemonic Name	Conventional Use	Register Number	Mnemonic Name	Conventional Use
$0	zero	Permanently 0	$24, $25	$t8, $t9	Temporary
$1	$at	Assembler Temporary (reserved)	$26, $27	$k0, $k1	Kernel (reserved for OS)
$2, $3	$v0, $v1	Value returned by a subroutine	$28	$gp	Global Pointer
$4–$7	$a0–$a3	Arguments to a subroutine	$29	$sp	Stack Pointer
$8–$15	$t0–$t7	Temporary (not preserved across a function call)	$30	$fp	Frame Pointer
$16–$23	$s0–$s7	Saved registers (preserved across a function call)	$31	$ra	Return Address

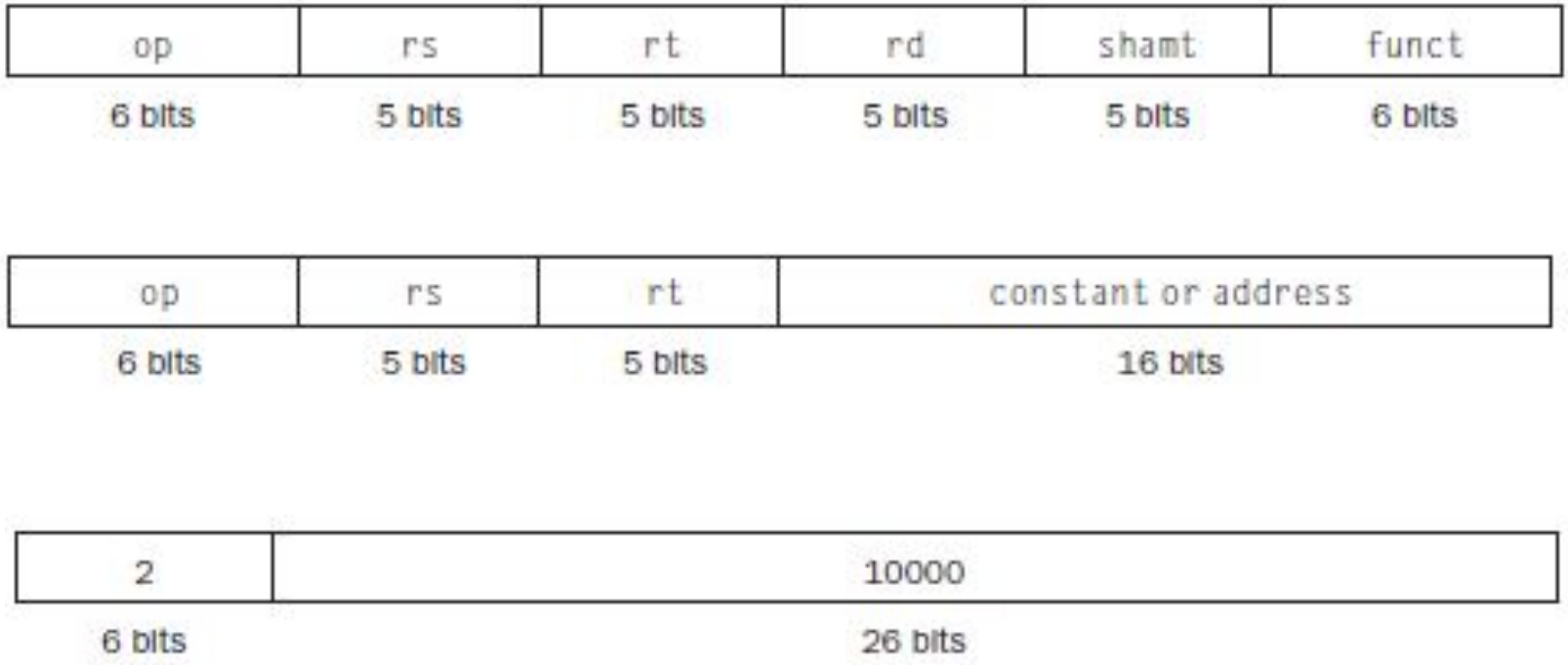
MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	$s1 = $s2 + $s3	Three register operands
	subtract	`sub $s1,$s2,$s3`	$s1 = $s2 – $s3	Three register operands
	add immediate	`addi $s1,$s2,20`	$s1 = $s2 + 20	Used to add constants
Data transfer	load word	`lw $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	`sw $s1,20($s2)`	Memory[$s2 + 20] = $s1	Word from register to memory
	load half	`lh $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	load half unsigned	`lhu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	`sh $s1,20($s2)`	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte	`lb $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	load byte unsigned	`lbu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	`sb $s1,20($s2)`	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	`ll $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	`sc $s1,20($s2)`	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	`lui $s1,20`	$s1 = 20 * 2^{16}	Loads constant in upper 16 bits
Logical	and	`and $s1,$s2,$s3`	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	`or $s1,$s2,$s3`	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	`nor $s1,$s2,$s3`	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	`andi $s1,$s2,20`	$s1 = $s2 & 20	Bit-by-bit AND reg with constant
	or immediate	`ori $s1,$s2,20`	$s1 = $s2 | 20	Bit-by-bit OR reg with constant
	shift left logical	`sll $s1,$s2,10`	$s1 = $s2 << 10	Shift left by constant
	shift right logical	`srl $s1,$s2,10`	$s1 = $s2 >> 10	Shift right by constant
Conditional branch	branch on equal	`beq $s1,$s2,25`	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1,$s2,25`	if ($s1!= $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for `beq`, `bne`
	set on less than unsigned	`sltu $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than unsigned
	set less than immediate	`slti $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant
	set less than immediate unsigned	`sltiu $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant unsigned
Unconditional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to $ra	For switch, procedure return
	jump and link	`jal 2500`	$ra = PC + 4; go to 10000	For procedure call

Writing Assembly — The Syntax of Instructions

- R-Type Instruction (Register)
  - Op: Operation (Opcode)
  - rs: first register source operand (destination)
  - rt: second register source operand (Operand 1)
  - rd: third register source operand (Operand 2)
  - shamt (the amount to shift bits by) (only in shift commands, otherwise it's 0)
  - func (function code for selecting variants of the operation, only when applicable)
- I-Type Instructions (Immediate)
  - Op
  - rs
  - rt
  - i (constant or address)
- J-Type Instructions (Jump)
  - Op
  - Address

Guess the instruction type?



Computation and Bitwise Logic

- All of our operations compute directly with binary data
- Logical (bitwise) Operations
  - sll, srl
  - and, andi
  - or, ori
  - nor
- Computational Operations
  - add, addi, sub, subi
  - mult, div

Writing MIPS Expressions

C Code	MIPS
`int x = y + 5;`	`addi $t0, $t1, 5`
`int x = y & z;`	`and $t0, $t1, $t2`
`int x = y | 5;`	`ori $t0, $t1, 5`
`int x = x << 7;`	`sll $t0, $t0, 7`
`some_function();`	`j some_function`

`if(x == y) {do_something}`
`else { ` ***`return;`*** ` }`
	`beq $t0, $t1, do_something`
	`addi $t0, $t1, 5`
	`j $ra`
	`do_something:`
	`sll $t0, $t0, 7`

Examples of code with demo instructions:
$t1 = 0000 0000
$t2 = 0000 0001

Instruction	$t0	$t1	$t2
srl $t0, $t2, 2	**0000 0100**	0000 0000	0000 0001
or $t1, $t0, $t2	0000 0100	**0000 0101**	0000 0001
sub $t2, $t1, $t0	0000 0100	0000 0101	**0000 0001**
and $t0, $t0, $t1	**0000 0100**	0000 0101	0000 0001

Hello World in Mips

```
# Hello, World!
        .data     ## Data declaration section
## String to be printed:
out_string:    .asciiz   "\nHello, World!\n"

        .text     ## Assembly language instructions go in text segment
main:             ## Start of code section
        li    $v0, 4            # system call code for printing string = 4
        la    $a0, out_string   # load address of string to be printed into $a0
        syscall                 # call operating system to perform operation
                                #   specified in $v0
                                # syscall takes its arguments from $a0, $a1, ...

        li    $v0, 10           # terminate program
        syscall
```

Service	System Call Code	Arguments	Result
print integer	1	$a0 = value	(none)
print float	2	$f12 = float value	(none)
print double	3	$f12 = double value	(none)
print string	4	$a0 = address of string	(none)
read integer	5	(none)	$v0 = value read
read float	6	(none)	$f0 = value read
read double	7	(none)	$f0 = value read
read string	8	$a0 = address where string to be stored $a1 = number of characters to read + 1	(none)
memory allocation	9	$a0 = number of bytes of storage desired	$v0 = address of block
exit (end of program)	10	(none)	(none)
print character	11	$a0 = integer	(none)
read character	12	(none)	char in $v0

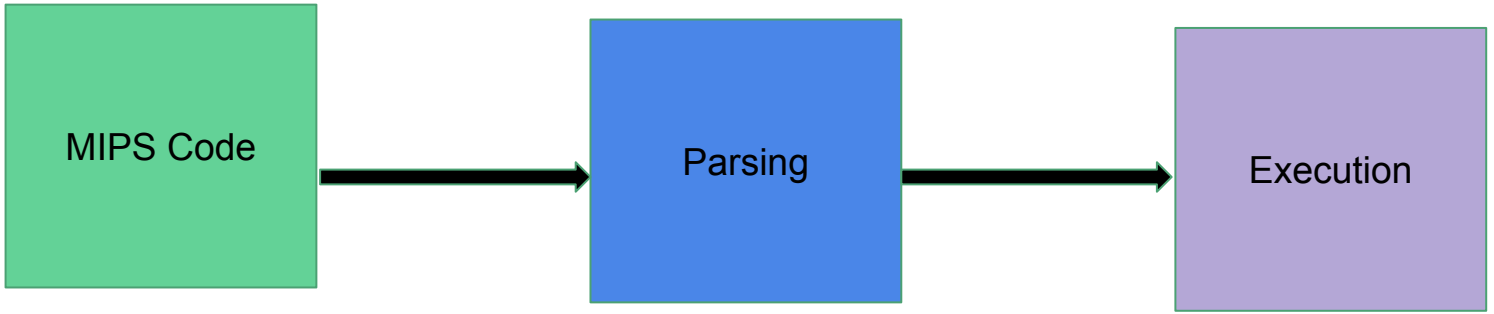
Takeaways from Hello World

- MIPS separates text and data segments in your file.
  - Text segment is for your code
  - Data segment is for “Static” memory, such as strings, single ints, or predeclared arrays
- Syscall is the way to tell the operating system what to do. We have to write some data into registers as “arguments” in order for syscall to do anything useful, such as printing to the console buffer.

What is an Interpreter?

- Many languages, including Python, use this method for running code.
- This approach differs from the typical compilation process, which looks at compiling your entire program at once.
- Rather than compiling our code into machine code, we interpret it using some black-box dynamically. This means our code never has to be stored in an executable. However, that means each time we run that we have to re-interpret.
- Interpreters have to look like by line at your code, and figure out how to perform them individually.
- *So, how about interpreting assembly with c++?*

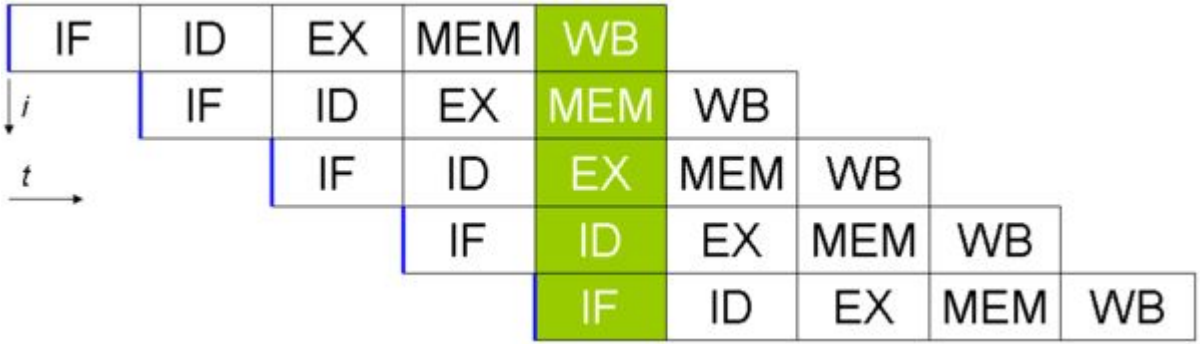
Basic Architecture of a MIPS Interpreter



With a Twist – The Pipeline

- The Pipeline relates our MIPS instructions to how our hardware processes them
- Our hardware components are synchronized by the computer clock
- In order for our computations to work in sequence, the MIPS designers utilized a “Pipeline” architecture to simplify sequencing multi-instruction execution
- Each instruction is processed in a specific hardware sequence:
  - IF ➔ ID ➔ EX ➔ MEM ➔ WB
  - IF = Instruction Fetch
  - ID = Instruction Decode
  - EX = Execute
  - MEM = Memory access
  - WB = Register write back

A Look at the classic RISC Pipeline



Simulating the Pipeline

- We can simulate the MIPS pipeline in our interpreter by doing a couple things:

1. As we read instructions (1 new per cycle), associate each instruction with a step in the pipeline
2. On ID, we know what registers the instruction will use. Check to make sure the registers used by the instruction are not being utilized by the previous instructions. This is called a “Data hazard.”

Ex: Notice we are using $t0 in the second instruction, but it will not be written to in time in the pipeline. Notice that the first instruction hasn’t performed WB on $t0 yet, but the second instruction is already executing. This is clearly a problem if we want our data to work sequentially!

1. `add **$t0**, $t1, $t2` IF… ID… EX… **MEM**… WB…
2. `add $t3, **$t0**, $t4` IF… ID… **EX**… MEM… WB…

The classic RISC solution

- Because we learn about hazard on ID (one stage away from EX), check the previous two instructions. This is because an instruction 3 cycles ahead (one processed 3 previously) will be on the WB stage, and will have written to the registers, so no need to check those.
- If we find that an instruction 1 or 2 before has a source/ destination register we are using in our current operation, we must “stall” the pipeline for our instruction.
- That means we stop on the ID stage and add NOP = “No Op” instructions to delay until the hazard clears.
- This means we must place our NOP inside the code sequence above our dangerous line of code to stall it.

Example of Stalling:

```
1.   add $t0, $t1, $t2     IF… ID… EX…
2.   add $t3, $t0, $t4          IF… ID…
```

In this moment it is clear we need to stall.

```
1.   add $t0, $t1, $t2     IF… ID… EX… MEM… WB…
2.   NOP                        IF  ID
3.   NOP                        IF  ID
4.   add $t3, $t0, $t4          IF… ID… EX…  MEM… WB…
```

Place two NOP instructions to give our original 2nd instruction enough stall time for the pipeline of instruction 1 to fully complete. Since a NOP has no real operation, we can say it will IF and ID simultaneously.

Is this really necessary?

- No, since when you write assembly it will be handled for you.
- C++ doesn’t need to care about this problem since it is much higher level.
- So, the point here is to simulate an interpreter that is actually coherent with the **architecture** of the hardware and, consequently, **instruction set**.

How does the Pipeline relate to our Parser?

Step	Consequence
IF	Do nothing.
ID	Check for hazards. Stall if needed.
EX	Do Nothing
MEM	Do Nothing
WB	Perform, or execute, the operation and write the result to the register. Do this here since there's no point calculating it on EX and waiting to write it on WB in C++.

The Relationship between Parser and Execution

- The parser reads the line of code and determines the instruction and any data, labels, or registers related to the problem.
- The executor only needs to know the memory location of the register and what data to write to it. Nonetheless, we still need a function for each command we want to implement since it varies depending on the operation.
- Note that at the hardware level, these would be done with multiplexers and ALU circuits to complete various operations. Also remember that a true MIPS parser would convert all op codes into numerical representation.

```
void add_(int* destRegister, int* leftRegister, int* rightRegister) {

    *destRegister = *leftRegister + *rightRegister;

}
```

Qtspim

- An open source MIPS simulator
- Mimics architecture of MIPS processor
- Download and install
- Write MIPS code in your favorite text editor
  - Recommended Notepad++ for simple code writing and text search and highlighting features
- Write code in .s files (source) indicating code written in assembly

MIPS code template

- Download this template
- Demonstrates:
  - Text and Data segments
  - String declaration
  - Array declaration
  - Loops
  - Printing

Problem Solving Lab

Please take time to work on solving the following 3 “Lab” problems. You should try working independently, but feel free to ask questions if you’re stuck. These problems are intended to be somewhat difficult, but you should have enough programming experience to have some intuitions about how to solve them.

At the end, I will share the solutions and we can discuss/ clarify any difficulties to solidify your understanding. I fully expect a variety of approaches, so please share if you find something clever :D

Lab: 3 Assembly Problems

For each of the following, print the solution for the problem. You do not need to take console inputs, so you can load whatever values you want into your registers to start.

1. Implement a two's complement subtractor (with only positive numbers) in MIPS so that you only use **add** and **nor** operations.
    a. Recall that two's complement involves flipping all bits (e.g. from 1 to 0 and 0 to 1) and then adding 1.
    b. You might be able to mimic subtracting by finding two's complement of the second number.
2. Determine whether a number in a register is even or odd in MIPS.
    a. Think about a numerical property of binary and a logic gate that relates to it.
3. Extract each of the four bytes from a given register in MIPS.
    a. Think about where any byte is stored in a word and a way to extract those bytes individually.

Some MIPS References to help

Reference sheet with instruction type

An easier reference sheet

My favorite reference

Assembly Problem Solutions