Project One – Single-Cycle and Pipelined CPU

Instructor: Dr. Anup Das

Project Overview

This project is intended to be a comprehensive overview on single-cycle MIPS CPU as well as an introduction to pipelined MIPS CPU. You may work on projects in group of two. There are three parts to this assignment.

1. In part one, you will study and test the single-cycle CPU codes we provided.
2. In part two, you will implement advanced instructions (stack operations) upon on the existing MIPS ISA to support real-life program executions. (8 points)
3. In part three, you will extend the single-cycle CPU to a pipelined CPU in order to complete project two. (12 points)

This project weights 20% of your entire letter grade while project two weights 30%. Both projects are newly designed and difficult, so please start as early as possible.

Contact

Project Designer: Shihao Song (ss3695@drexel.edu)

Office/Hours: Bossone 616/T 10AM to 11AM; TH 3PM to 4PM

Notes: Questions such as VHDL/C/C++/Modelsim tutorials, code debugging are not acceptable in 400/600-level courses.

Level of Difficulty

Project	Level of Difficulty
ECEC 355 Quarter-long Project	1
ECEC 412 Project One - Part Two	2
ECEC 412 Project One - Part Three	3
ECEC 412 Project Two - Part One	3
ECEC 412 Project Two - Part Two	3.5
ECEC 412 Project Two - Part Three	3

Part One and Part Two – Stack Operations

In part one, please read through the codes we provided and get yourself familiar with what you learnt in ECEC 355. I won't grade part one, however, I highly recommend you do the following things:

(1) Hand-write some assembly codes in MIPS and convert them into machine codes, please read chapter 2 of your ECEC 355 textbook with the MIPS Green Card provided if you forgot how to do it.

(2) Simulate the single-cycle MIPS CPU with the machines codes you wrote in (1) and make sure the single-cycle CPU works as expected.

In part two, you will implement instructions on stack operations to the existing single-cycle MIPS CPU circuit. But before we start, let's do an overview on stack.

```
void hello_world(int num)
{
    int new_num = num + 1;

    cout << "The number you entered is " << new_num << endl;
}

int main()
{
    static int global = 5;

    hello_world(3);

    return 0;
}
```

Figure 1 Simple C++ program

As shown in Figure 1, hello_world() is called inside the main function, then a local variable called "new_num" is created and printed out. So what is really happening (CPU level) when hello_world() is invoked?

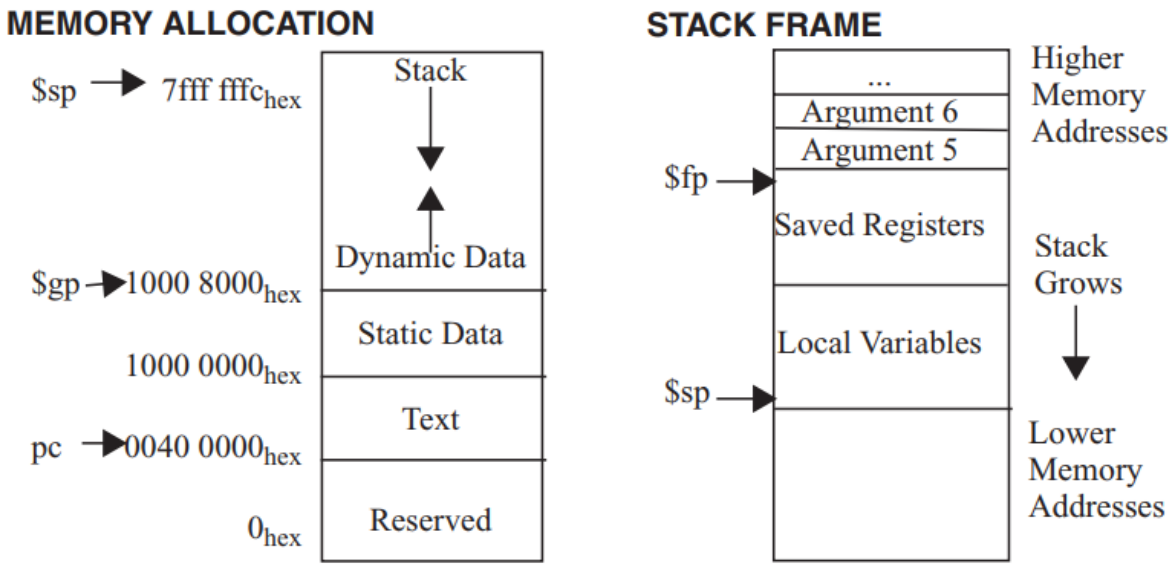

Figure 2 Memory Layout

When the C++ program is compiled into assembly language (the assembly language codes are usually inside a temporary file with extension ".out") and brought into memory (should be virtual memory more precisely) to be executed, the program is actually being "sectioned" inside memory (Figure 2 left). As we can see from Figure 1, "return 0", "cout << ", "int new_num"… They are the codes, thereby, those codes will be put into section Text (Figure 2 left). Meanwhile, the "static int global 5" will be put into section Static (Figure 2 left). You may wonder that it seems that "int new_num" should also be put into Static section since it is data. Well, "new_num" is a local variable inside hello_world(), it is only used when hello_word() is running and destroyed when program counter returns back to main function. So, the compiler and CPU simply treat local variables as run-time members instead of permanent members, we will discuss what it means later.

Now, let's assume the program counter points to line of hello_world(3) in main function. Initially, there is nothing inside stack and stack pointer is at its initial position which is usually the highest memory address as shown below.

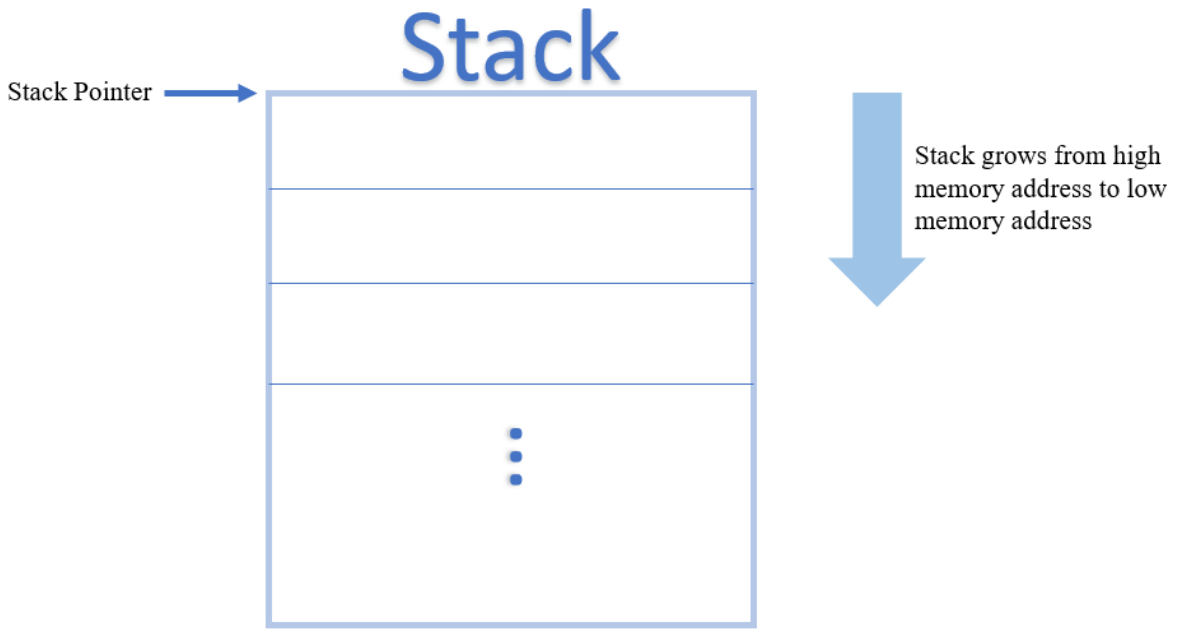

Figure 3 Before hello_world() is invoked

Next, the argument 3 along with other system-level status info (this is where most hackers take advantage of but we won't cover it in this course) will be pushed into stack.

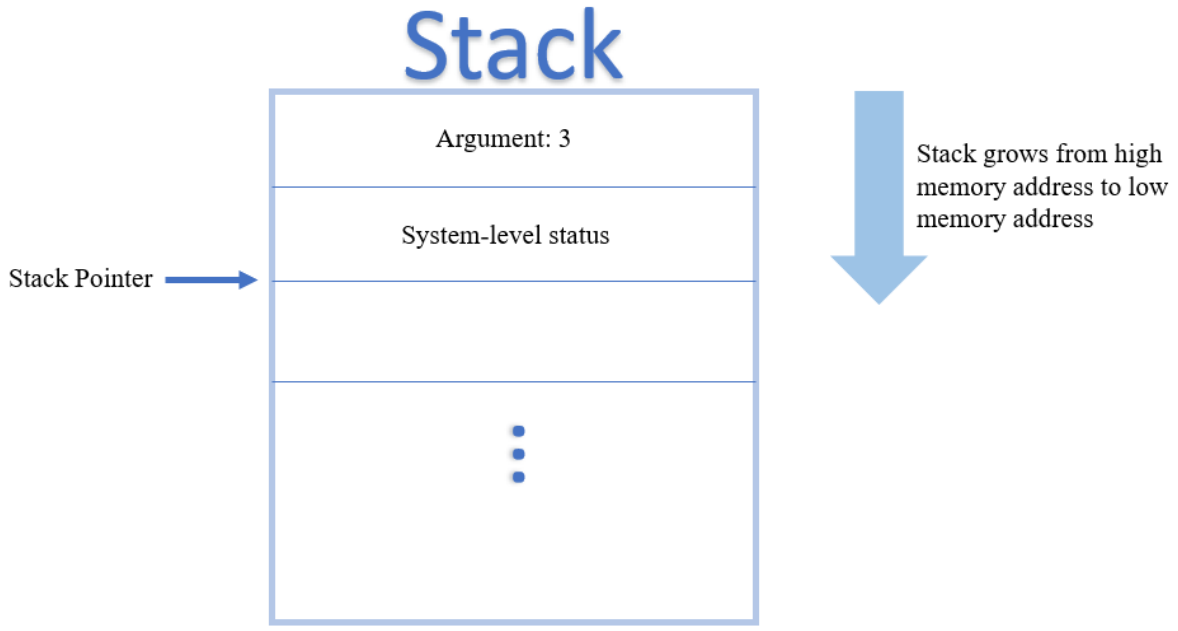

Figure 4 Ready for execution

At this point, the function hello_world() is ready for execution. Within its execution period, the local variable "new_num" will be created (pushed) in stack.

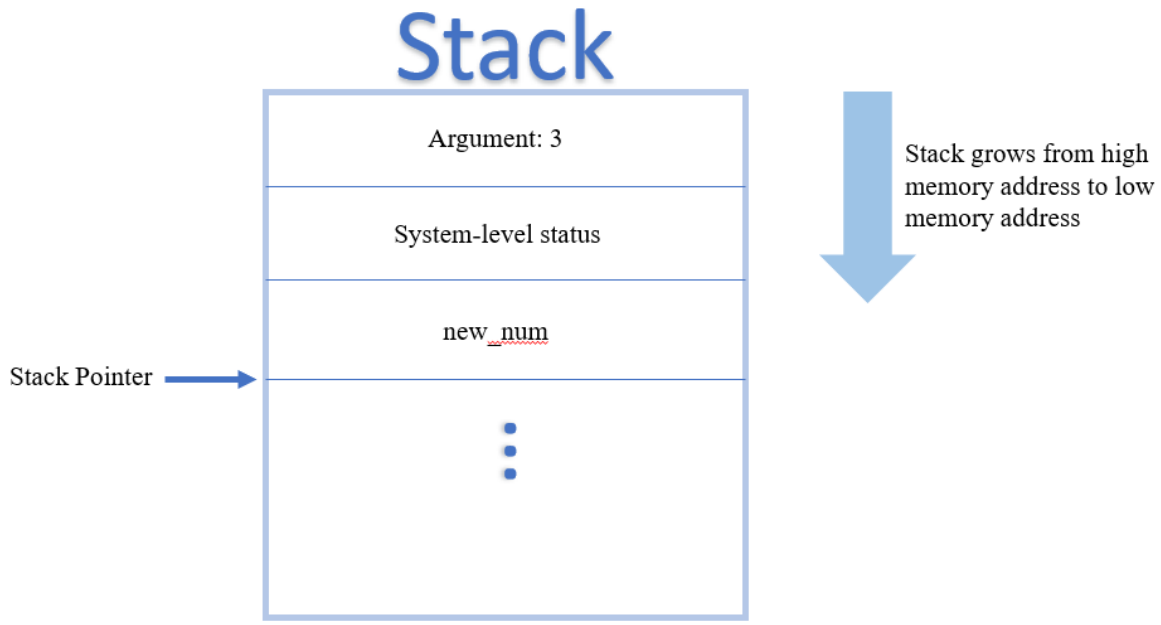

Figure 5 At execution

When the execution completes (program counter returns back to main function), local variables should be popped out of stack (something else will happen but let's just assume only local variables get popped out).

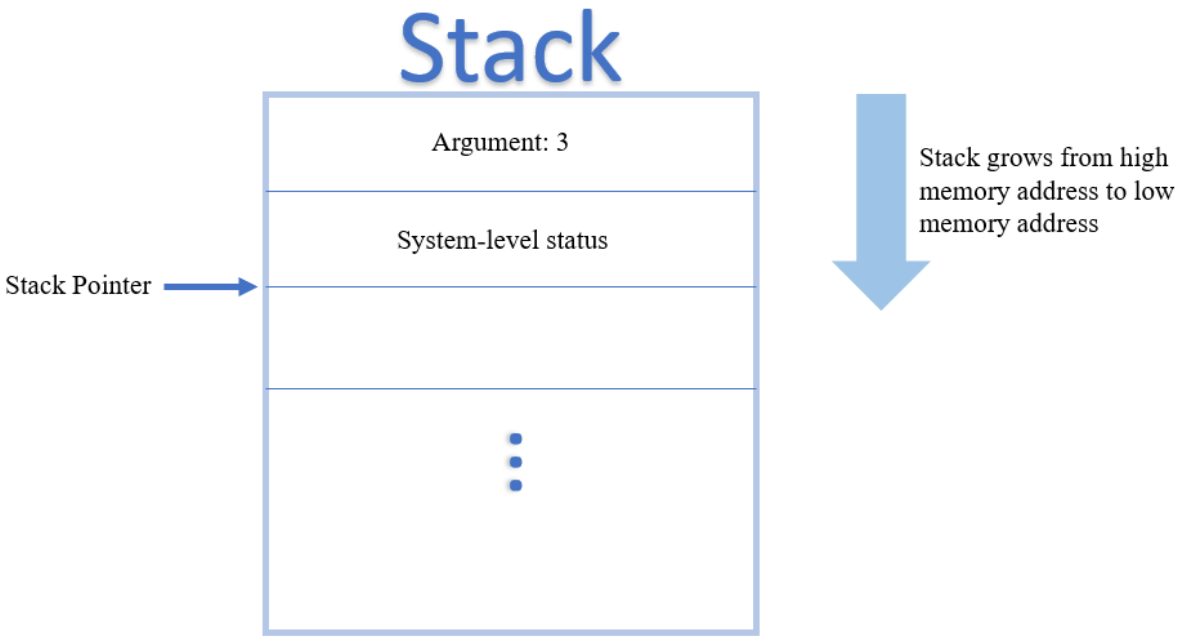

Figure 6 After execution, local variable got popped out.

What you need to do

As we can see from the MIPS Green Card, push and pop are not there… So what you need to do is to design your own push and pop instructions for MIPS. Finally, your single-cycle MIPS should support instructions such as:

push 4 (push number 4 into stack)

pop $s1 (pop the word into register $s1)

Steps

1. Let's assume push and pop are both I-type instructions; MIPS Green Card does not provide the OPCODE/FUNCT for push and pop, so you are free to make them up unless they don't affect the existing opcodes inside control unit.
2. You should add one more register (stack pointer) to your register file.
3. As for the stack, you can just treat the data cache as a stack and initialize the stack pointer to the highest location of data cache.
4. Consider what is behind push and pop?

   Hint: push 4 = [sub $sp, 4] + [sw 4, 0($sp)]; pop $s1 = [lw $s1, 0($sp)] + [add $sp, 4].

   You may need to add additional wires, mux or control signals, anyway, I leave it for you to figure it out.

Check-off

Just show me the simulations of the followings:

push 4 (After one clock cycle, I should see 4 in the stack)

pop $s1 (Similarly, after one clock cycle, I should see 4 inside $s1)

Part Three

Make a copy of the project you have done so far, name the new project as “pipeline_mips”. Before we move on to pipelined CPU, I want you to logically separate your data cache into two parts: stack (upper half of your data cache) and data section (lower half of your data cache). In order to do this, you need to add one more register to your register file: $dp (data pointer), initialize it to the lowest address (0) of data cache.

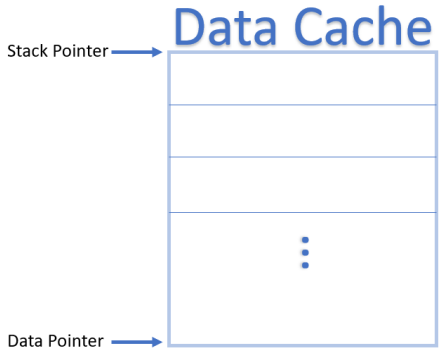

Figure 7 Data Cache

Finally, extend the existing single-cycle CPU to a pipelined CPU given in Figure 4.49 of Project One Supplement.

Initialization

```
regcontents(0)<=(others=>'0');
regcontents(8)<=(others=>'0');
regcontents(9)<="00000000000000000000000000000100";
regcontents(10)<="00000000000000000000000000000100";
regcontents(20)<="00000000000000000000000000001110";
regcontents(21)<="00000000000000000000000000000101";
regcontents(22)<="00000000000000000000000000001000";
regcontents(23)<="00000000000000000000000000000011";

memcontents(0)<="00000000";
memcontents(1)<="00000000";
memcontents(2)<="00000000";
```

```
memcontents(3)<="00000100";
memcontents(4)<="00000000";
memcontents(5)<="00000000";
memcontents(6)<="00000000";
memcontents(7)<="00000100";
```

Check-off

Show me the simulations of the following codes:

Code One (No hazards)

```
add $s3,$s4,$s5
lw $s0,4($dp)
lw $s1,0($dp)
sub $s7,$s4,$s6
sw $s3,8($dp)
```

Code Two (First, detect hazards and add NOPs to overcome the hazards; second, simulate the code with NOPs)

```
add $s3,$s4,$s5
lw $s0,0($dp)
lw $s1,4($dp)
sub $s7,$s4,$s6
sw $s7,8($dp)
```

Code Three (First, detect hazards and add NOPs to overcome the hazards; second, simulate the code with NOPs)

```
push 8
add $s3,$s4,$s5
lw $s0,0($dp)
lw $s1,4($dp)
pop $t0
```

```
add $s7,$t0,$s1
sw $s7,8($dp)
```