CS 252 (Spring 17): Computer Organization

# Project #11

Pipelined CPU

due at 5pm, Wed 26 Apr 2017

## 1 Purpose

In this project, you will be adapting your Project 9 code to build a pipelined processor. This processor will support the same instructions as Project 9 (plus a few more). My testcases will provide the IF phase; you will implement the other 4 pipeline phases. My code will call your 4 functions at the proper times.

For each phase, you will write an `execute_*` function; the parameters will typically be the input pipeline register (the one to the left) and the output pipeline register (the one to the right). You must **never** modify the input register - and you must set all of the fields of the output register, every time.

To implement branches, jumps, and stalls, the `execute_ID()` function has a second output parameter: the `ID_branchControl` struct. This struct, described below, has fields which allow the ID phase to either stall the processor, or to branch to a new PC.[1]

Your processor will implement data forwarding; this means that the `execute_EX()` function will include additional parameters: the current `EX/MEM` and `MEM/WB` registers, so that you can forward values from there.[2]

Your processor will also implement `LW` stalls, and so `execute_ID` will include a pointer to the current `ID/EX` register.

Two functions will be passed a pointer to the `regs[]` array. `execute_ID()` must read from this array, but **must not** modify it (even for the `jr` instruction). At the other end, `execute_WB` will also have a pointer to the registers array - which it will use if it needs to write to a register.[3]

Likewise, the `execute_MEM()` function has a parameter which points to the data memory.

### 1.1 Required Filenames to Turn in

Name your C file `proj11.c`. No README is required this time.

[1] Our processor will **not** implement the Branch Delay Slot (even though real MIPS processors do that). Instead, if you decide to branch, that new instruction will show up in the very next clock cycle. In other words, this processor is simpler than the real hardware.

[2] You may have noticed that the `BEQ/BNE` instructions needs data forwarding as well, but that must happen in the ID phase. To simplify our processor, we'll ignore this - I have inserted enough `NOP` instructions, before each branch, to eliminate this worry.

[3] This project does not require that you implement any extra instructions - but if you want to do so, you may. To that end, the `regs[]` array is 34 words long, instead of 32 - to account for the `lo/hi` registers used by `mult` and `div`.

### 1.2 More Complex Testcases

Because the testcases have a lot of C code that they share, I now have some "common code" which is shared across all of them. This is contained in a header file `proj11_test_commonCode.h` and the matching source file `proj11_test_commonCode.c`. Each testcase will include the header, and link with the shared code.

While you are allowed to include the test-common-code header if you want, I don't think that you will need it. Feel free to poke around the code, however.

## 2 Instructions

Every student's code will be required to suppor the same set of basic instructions.

### 2.1 Required Instructions

Your CPU must support all of the following instructions. Instructions that were not required in Project 9 are highlighted in **red.**

As before, treat all 'u' instructions exactly the same as their more ordinary counterparts. Use C addition and subtraction, using signed integers, and ignore overflow.

- `add, addu, sub, subu, addi, addiu`
- `and, or,` **andi, ori**
- `slt, slti,` **sltu, sltiu**
- `lw, sw`
- `beq,` **bne,** `j,` **jal, jr**

The testcases will handle the `syscall` instruction on your behalf - you don't have to write any code to make it work.[4]

### 2.2 Additional Instructions Not Required

Unlike Project 9, this project does not require that you support any extra instructions. However, if you want to, I have added 3 'extra' fields to each pipeline register. I have also increased the number of registers from 32 to 34, to support the `lo/hi` registers used by `mult` and `div`.

[4] `syscall` is another instruction with data-forwarding problems. In a real processor, `syscall` is actually a jump into code provided by the operating system, and so data forwarding is not a worry; it's just ordinary code. But in our little simulation, I insert `NOP`s so that the syscall will see the proper values for `$v0,$a0` when it runs.

### 2.3 Unsupported Instructions

If you are asked to decode an instruction which has an opcode or opcode/funct which you do not support, then you must return nonzero from `execute_ID()`.

## 3 Structs (that were not in Project 9)

I have provided `proj11.h`, which defines several types for you (you can't change them), and the prototypes for each of the functions which you must implement.

### 3.1 Pipeline Register Structs

Each pipeline register has a matching struct. The fields are well documented in `proj11.h`, but you will find that they are very similar to the fields from the various structs in Project 9.

### 3.2 `struct ID_branchControl`

This is the second output struct for `execute_ID()`. It has three fields. If all three are 0, then the testcase will simply advance you to `PC+4`, as normal. However, if you set the fields of this struct, you can force the testcase to handle other situations:

- `stallIF` (1 bit) - If you set this to 1, then on the next clock cycle, the ID phase will see **exactly** the same instruction (at the same PC).

  This bit does not change how the pipeline treats the ID/EX pipeline register, however. Thus, if `execute_ID()` wants to stall, it must set this bit **and** set the ID/EX pipeline register to NOP.

- `branch` (1 bit) - If you set this bit to 1, then the processor will jump to the PC given in the `branchDest` field, starting on the next clock cycle. Again, `execute_ID()` is responsible for setting a NOP in the ID/EX register.

  **NOTE:** For simplicity, our processor does **not** implement a branch delay slot. So you will see the new PC immediately - that is, on the very next clock cycle.

- `branchDest` (32 bits) - The PC to jump to if `branch=1`. `execute_ID()` should set it to zero otherwise.

## 4 `execute_ID()`

This function reads from the IF/ID pipeline register, decodes the instruction, and sets the fields of the ID/EX pipeline register. It must also set the `ID_branchControl` struct.

This function should be nearly identical to your `execute_CPUControl()` from Project 9. The fields of the ID/EX pipeline register are (mostly) the

same as the fields of the `CPUControl` struct. (See `proj11.h` for description of the changes.)

## 4.1 Updated `ALUsrc` Field

In Project 9, it was possible to implement zero-extended immediate values, but it required use of one of the extra fields. However, in Project 11, we have updated the meaning of `ALUsrc` to support this. (In order to match the testcases, you **must** use the new design.)

The possible values for `ALUsrc` are now:

- 0 - use the value of the rt register
- 1 - use the sign-extended immediate field
- 2 - use the **zero**-extended immediate field

## 4.2 Branch Control

`execute_ID()` must implement both `BEQ` and `BNE`; set the `ID_branchControl` struct to indicate whether a branch should happen. Likewise, `execute_ID()` must implement `j, jal,` and `jr`.

Remember that, even if you indicate that a branch should occur, you must still set the ID/EX pipeline register; typically, this will be to force a NOP for the current cycle. However, `jal` is different.

`execute_ID()` **must not** modify any registers. Thus, `jal` cannot be entirely executed in the ID phase. Instead, use the normal operations of the ALU to set `$ra`. While there are lots of ways to do this, you need to match the expected output, and so your code must perform a modified `ADDI` instruction. (See the testcase output to see the exact control bits.)

## 4.3 `LW` Hazard Stalls

One of the parameters to `execute_ID()` is the old (that is, the "before") contents of the ID/EX register. This, of course, is the input to the EX phase this clock cycle.

`execute_ID()` must detect when a `LW` instruction is in the EX phase, and it writes to one of the two registers being read in the ID phase. If this condition exists, the ID phase must stall.

When doing this check do **not** check the opcode. Instead, simply read the `rs,rt` fields from the instruction (even if they are not going to be used, or if the instruction writes to rt). If the instruction in EX is a `LW`, and it writes to either of these registers, then assume that a hazard exists.

### 4.4 Return Value

If there was an error (unrecognized opcode or funct), then return nonzero from this function. Return 0 if the function completed normally. (Note that a branch or stall is **not** an error condition!)

## 5 `execute_EX()`

This function implements the EX phase. It should basically be the combination of your `execute_ALU()` and `getALUinput2()` functions from Project 9.

However, your code will need to be improved a little bit: this function must implement data forwarding (which is why it has pointers to the current EX/MEM and MEM/WB pipeline registers).

In addition, this function must now support the expanded `ALUsrc` field (see above).

Finally, this function must choose the destination register: it will read the `regDest, rt, rd` fields from ID/EX and set the `destReg` in EX/MEM.

## 6 `execute_MEM()`

This function will be very similar to `execute_MEM()` from Project 9.

## 7 `execute_WB()`

This function will be very similar to `execute_updateRegs()` from Project 9.

# 8 A Note About Grading

Your code will be tested automatically. Therefore, your code must:

- Use exactly the filenames that we specify (remember that names are case sensitive).
- **Not** use any other files (unless allowed by the project spec) - since our grading script won't know to use them.
- Follow the spec precisely (don't change any names, or edit the files I give you, unless the spec says to do so).
- (In projects that require output) match the required output **exactly!** Any extra spaces, blank lines misspelled words, etc. will cause the testcase to fail.

To make it easy to check, I have provided the grading script. I **strongly recommend** that you download the grading script and all of the testcases, and use them to test your code from the beginning. You want to detect any problems early on!

### 8.1 Testcases

You can find a set of testcases for this project at
`http://www.cs.arizona.edu/classes/cs252/spring17/projects/proj11/`
You can also find them on Lectura at
`/cs/www/classes/cs252/spring17/projects/proj11/`
(this is the backing directory for the website).

For assembly language programs, the testcases will be named `test_*.s` . For C programs, the testcases will be named `test_*.c` . For Java programs, the testcases will be named `Test_*.java` . (You will only have testcases for the languages that you have to actually write for each project, of course.)

Each testcase has a matching output file, which ends in `.out`; our grading script needs to have both files available in order to test your code.

For many projects, we will have "secret testcases," which are additional testcases that we do not publish until after the solutions have been posted. These may cover corner cases not covered by the basic testcase, or may simply provide additional testing. **You are encouraged to write testcases of your own, in order to better test your code.**

### 8.2 Automatic Testing

We have provided a testing script (in the same directory), named `grade_proj11`. Place this script, all of the testcase files (including their `.out` files if assembly language), and your program files in the same directory. (I recommend that you do this on Lectura, or a similar department machine. It **might** also work on your Mac, but no promises!)

### 8.3 Writing Your Own Testcases

The grading script will grade your code based on the testcases it finds in the current directory. Start with the testcases I provide - however, I encourage you to write your own as well. If you write your own, simply name your testcases using the same pattern as mine, and the grading script will pick them up.

While you normally cannot share code with friends and classmates, **testcases are the exception.** We encourage you to share you testcases - ideally by posting them on Piazza. Sometimes, I may even pick your testcase up to be part of the official set, when I do the grading!

## 9 Turning in Your Solution

You must turn in your code using D2L, using the Assignment folder for this project. Turn in only your program; do not turn in any testcases or other files.