CS 252: Computer Organization

Simulation #5

Pipelined CPU

due at 3pm, Sat 01 May 2021

1 Purpose

In this project, you will be adapting your Simulation 4 code to build a pipelined processor. Each pipeline register will be represented by its own struct; for instance, the ID/EX pipeline register is represented by the struct `ID_EX`. Just like in a real processor, each pipeline register must contain all of the information that is saved from one clock cycle to another; there will **not** be any global Control struct (as there was in Simulation Project 4).

The core of each pipeline phase is an `execute_*` function, which typically will read from the pipeline register on the left, and write to the pipeline register on the right. However, the details vary from phase to phase - so read the spec carefully.

Sim 5 will be a lot easier to write if you have Sim 4 to base it off of. So, if you didn't complete that project, your first step should be to finish it!

1.1 Required Filenames to Turn in

Name your C file `sim5.c`.

2 Required Instructions

Every student must implement the following instructions. (The instructions in red will require you to expand the standard CPU design by adding extra control bits, or extra values to existing controls.)

- `add, addu, sub, subu, addi, addiu`

  (Treat the 'u' instructions exactly the same as their more ordinary counterparts. Just use C addition and subtraction, using signed integers. Ignore overflow.)

- `and, or,` **xor, nor**
- `slt, slti`
- `lw, sw`
- `beq, bne, j`
- **andi, ori**
- **lui**

- **nop**

  Remember that the instruction `NOP` is all zeroes, which works out to be `sll $zero,$zero,0`. You may implement the full SLL instruction if you wish - but the only thing that you are **required** to do is to make sure that a NOP instruction will work. (See below for some details about the required ALU op.)

2.1 No Extra Instruction Requirement

Unlike Simulation 4, I am not requiring that you implement any extra instructions, beyond the list above. However, I've tried to implement the project in such a way that it is **possible** to implement more, if you find that interesting.

2.2 Extended Control Values

Just like in Simulation 4, we'll need to modify the standard CPU design a bit. You **must** add the following features:

- Set `ALUsrc=2` to indicate **zero-extended imm16**
- Set `ALU.op=4` to select the XOR operation in the ALU
- Set `ALU.op=5` if the operation is a `NOP`

  (If you wish to do a full SLL implementation, that's fine - but your design **must** implement at least `NOP`, and it must use `ALU.op=5` when it does.)

(This will be enough to pass the testcases that print out lots of debug data.)

In addition, as noted above, you must support other instructions as well - but for those, you can decide how to implement them. While you can expect that I will test them, I won't print out debug data when I do.

3 Data Forwarding

In class, we've talked about two types of data forwarding: forwarding into the ALU, and forwarding into the "Write data" port of the memory unit. In this project, you will implement both.

Forwarding into the ALU is handled through the `EX_ALUinput*` functions. You will read control wires (both from your own pipeline register, and also from the EX/MEM and MEM/WB pipeline registers), and return the correct value. Of course, you will need to do this for both inputs.

In addition, you will implement forwarding into the "Write data" port of the memory unit. The only time that this happens is when the **data** register that you want to write was modified by the instruction just **before** the `SW` instruction.

3.1 Things to think about for the future...

There are some other possible places where you could forward in the MIPS pipeline, which we haven't discussed in class. You can simply **ignore** these for your project (my testcases won't test these). However, I wanted to include them for completeness:

- `BEQ,BNE`

  In our updated design, we do register compares in the ID phase (not in the ALU). Thus, our ALU forwarding logic doesn't help us! So, technically, you really ought to stall the CPU if either of the two registers are being modified are still in the pipeline.

- `JR`

  This project doesn't include the `JR` instruction - but it would have the same issue as `BEQ,BNE` if it did!

- `SYSCALL`

  In a real processor, a `SYSCALL` is basically a type of jump (to a fixed address), and so forwarding isn't an issue. And once the jump takes place, you are running ordinary MIPS code (but inside the operating system), so ordinary forwarding works well.

  But in our simulation, I have special code to handle `SYSCALL`; it's almost like a magical, very powerful instruction. And the first step is that I read `$v0,$a0`.

  To solve any worries about forwarding, you will notice that I always insert `NOP`s into my code, before I call `SYSCALL` - at least, if there is a hazard to resolve.

4 Stalls

In this project, my code will ask your code **whether** a stall is required. You must detect the conditions listed below, and request a stall if they occur. (Your stall detection code always runs as part of ID.)

My code will handle the IF phase; if you ask for a stall, then I will stall the IF as well. However, it will be your responsibility to make sure that the control bits (in the ID/EX pipeline register) are all set to zero.

You must detect the following conditions:

- `LW` **Data Hazard**

  We've discussed this in class: one instruction loads a value from memory into a register, and the very next instruction wants to use that value in the ALU. Stall the 2nd instruction, to insert an empty space between them.

  This check must be **precise.** This means that you must look at the opcode, and figure out exactly which registers you are actually reading.

Only stall if a **real hazard** exists. (For instance, ADDI should not stall if the "hazard" only affects the rt register, since ADDI uses rt as a write register, not a read register.)

- `SW` **Data Register Hazard**

  This hazard occurs when a SW instruction plans to write a register to memory, but the **data** register that it plans to write is being modified by an instruction still in the pipeline.

  There are two versions of this. If the instruction that writes to the register is **immediately** before the `SW`, then you are required to solve this with forwarding.

  However, if the instruction that writes to the register is 2 clock cycles ahead, then you can't solve the problem with forwarding (unless we add even more complexity to the processor), so in that case you must stall[1]

  Bonus question: Why don't you need to stall if there is a write which is 3 clock cycles ahead???

5 Branches

Your CPU will implement conditional and unconditional branches. These will be resolved in the ID phase; you will indicate when/if a branch is required, as well as the destination of the branch (see the details below).

Of course, in a real CPU, there would already be an instruction in the IF phase when a branch occurs. Two strategies are classically used; either the instruction in IF is flushed (causing a NOP in the instruction stream), or else a **branch delay slot** is used.

In our CPU simulation, we'll use a third, simpler method - it isn't technically correct, but it's easy to understand: we will "magically" fetch the next instruction (at the branch destination), and have it ready for execution in the very next clock cycle.

However, you will be required to handle the ID/EX pipeline register. Since you will be handling all of the branch logic in ID, this means that every branch instruction (conditional or unconditional) must write all zeroes to the ID/EX pipeline register[2].

6 The Phases

Generally, each phase reads from the pipeline register on the left, and writes to the pipeline register on the right. The testcase will typically have two copies

[1] Sorry, you can't make your processor smarter than this. If you do, you won't pass the testcase.

[2] If you choose to go beyond the required project, and implement the `JAL` instruction, then you will need to do some sort of operation for `JAL`. But this is not necessary for any of the required instructions.

of each pipeline register: the "old" and "new" values. The old values are the contents of the register at the beginning of this clock cycle, and the new values are the contents of the register at the end. Thus, you may write to the "new" copy of each register, without worrying that you might be overwriting critical data in the "old," which might be needed by another pipeline phase during this cycle.

6.1 IF

The IF phase will be implemented by my testcases - however, your code in ID will instruct it what to do.

6.2 ID

The ID phase is the most complex of all, because it has to implement both stall and branch/jump logic. For this reason, there are several different function calls which will occur:

- `extract_instructionFields()`

  This works exactly like Simulation 4.

- `IDtoIF_get_stall()`

  This function asks if a stall is required. If you need to stall the ID phase, return 1; if not, then return 0.

  The parameters are the Fields of the current instruction, plus the ID/EX and EX/MEM pipeline registers, for the two instructions ahead. This function **must not modify any of these fields** - simply query to determine if a stall is required.

  If a stall is required, then the IF phase will also stall; that is, you will see this instruction repeated on the next clock cycle.

  If you don't recognize the opcode or funct (meaning that this is an invalid instruction), return 0 from this function.

- `IDtoIF_get_branchControl()`

  This asks the ID phase if the current instruction (in ID) needs to perform a branch/jump. The parameters are the Fields for this instruction, along with the rsVal and rtVal for this instruction.

  If you return 0, then the PC will advance as normal. (If you ask for a stall, then you must also return this branchControl value.)

  If you return 1, then the PC will jump to the (relative) branch destination - see `calc_branchAddr()`.

  If you return 2, then the PC will jump to the (absolute) jump destination - see `calc_jumpAddr()`.

If you return 3, then the PC will jump to rsVal. (You will not need to use this feature unless you decide to add support for `JR`, just for fun.)

If you don't recognize the opcode or funct (meaning that this is an invalid instruction), return 0 from this function.

- `calc_branchAddr()`

  This asks you to calculate the address that you would jump to if you perform a conditional branch (`BEQ,BNE`). This function should model a simple branch adder in hardware - and thus, it will calculate this value on **every clock cycle, and for every instruction** - even if there is no possible way that it might be used.

  Essentially, this is one of 4 inputs to a MUX - where the MUX is controlled by the branchControl value above. (My code in the IF phase will implement this MUX.)

- `calc_jumpAddr()`

  This asks you to calculate the address that you would jump to if you perform an unconditional branch (`J`[3]).

  As with `calc_branchAddr()`, this must calculate this value on every clock cycle, and for every instruction - even if there is no possible way that it might be used.

- `execute_ID()`

  This function implements the core of the ID phase. Its first parameter is the `stall` setting (exactly what you returned from `IDtoIF_get_stall()`). The next is the Fields for this instruction, followed by the rsVal and rtVal; last is a pointer to the (new) ID/EX pipeline register.

  Decode the opcode and funct, and set all of the fields of the `ID_EX` struct. (I don't define how you might use the `extra*` fields.)

  As in Simulation 4, you will return 1 if you recognize the opcode/funct; return 0 if it is an invalid instruction.

6.3 EX

The EX phase includes the following functions:

- `EX_getALUinput1()`

  This function must return the value which should be delivered to input 1 of the ALU. The first parameter is the current ID/EX register; it also has pointers to the current EX/MEM and MEM/WB registers.

  This function must not modify any of the structs; it just returns a 32-bit value.

[3] This would also be used for the `JAL` instruction, if you decided to add support for that.

- `EX_getALUinput2()`

  This is the same function, but for ALU input 2. It must also handle immediate values (see the `ALUsrc` control value).

- `execute_EX()`

  This function implements the core of the EX phase. It has a pointer to the ID/EX pipeline register (which it must not modify), the two input values (see above), and a pointer to the EX/MEM pipeline register (which it must fill).

  This phase must choose between the two possible destination registers (`rt,rd`); store the chosen register into `writeReg` (a 5-bit field). Notice that the `regWrite` control (1 bit) indicates **whether** we will write to a register, and `writeReg` (5 bits) indicates **which** we will write to.

6.4 MEM

The MEM phase only includes a single function:

- `execute_MEM()`

  This function works more or less like `execute_MEM()` from Simulation 4; it may read or write memory. The only new feature is that you must handle SW data forwarding - that is, the value that you write to memory might come from the MEM/WB register ahead of you.

6.5 WB

The WB phase only includes a single function:

- `execute_WB()`

  This function works more or less like `execute_updateRegs()` from Simulation 4; it may update a register.

7 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!

7.1 Testcases

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.**

7.2 Automatic Testing

We have provided a testing script (in the same directory), named `grade_sim5`. 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!)

7.3 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!

8 Turning in Your Solution

You must turn in your code using Gradescope, Turn in only your program; do not turn in any testcases or other files.