LSU EE 4720 **Homework 3** **Due: 7 March 2025**

Solution *Formatted 18:16, 21 March 2025*

Collaboration Rules

Each student is expected to complete his or her own assignment. It is okay to work with other students and to ask questions in order to get ideas on how to solve the problems or how to overcome some obstacle (be it a question of MIPS or assembler syntax, interpreting error messages, how a part of the problem might be solved, etc.) It is also acceptable to seek out assembly language resources for help on MIPS, etc. It is okay to make use of AI LLM tools such as ChatGPT and Copilot to generate sample code. (Do not assume LLM output is correct. Treat LLM output the same way one might treat legal advice given by a lawyer character in a movie: it may sound impressive, but it can range from sage advice to utter nonsense.)

After availing oneself to these resources **each student is expected to be able to complete the assignment alone**. Test questions will be based on homework questions and **the assumed time needed to complete the question will be for a student who had solved the homework assignment on which it was based**.

Student Expectations

Some of the problems require thought, and students are expected to persevere until they find a solution. It is each student's duty to him or herself to resolve frustrations and roadblocks quickly, hopefully helped along by the satisfaction of making progress. There are plenty of old problems and solutions to look at. One way to resolve issues is to ask Dr. Koppelman or others for help.

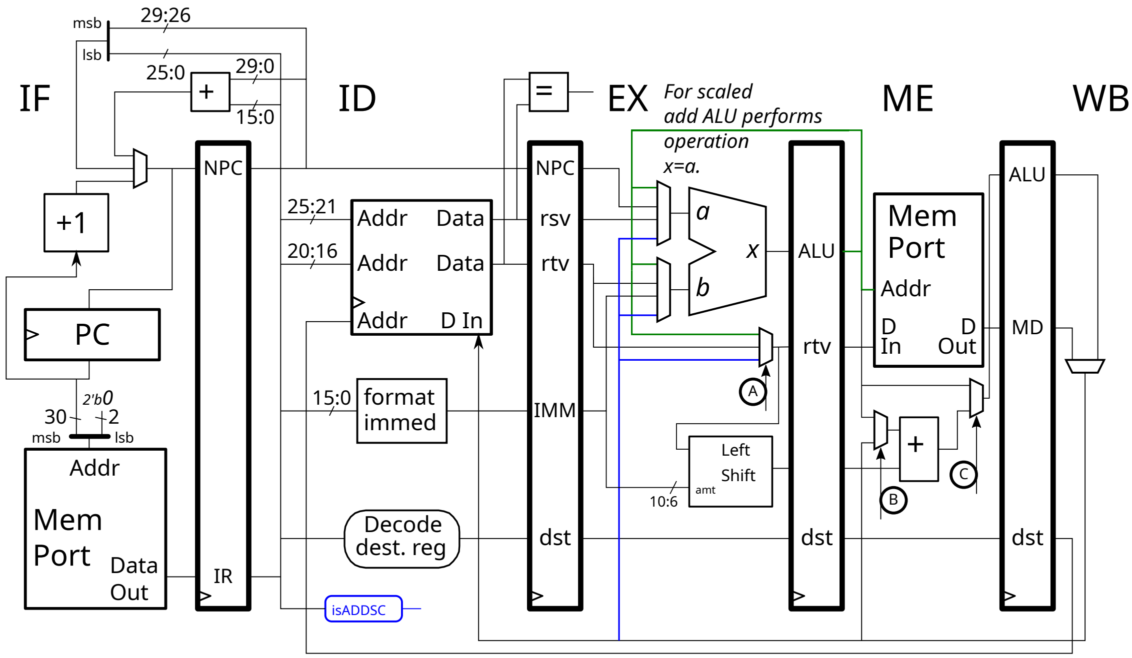
Resources

For examples of pipeline execution diagrams of given code fragments running on given MIPS implementations see past midterm exams (and final exams, but mostly midterms). The solutions to almost all past midterms in this course are available. A good place to start would be 2023 Midterm Exam Problem 2, 3, 4, and 5.

Homework Background

This assignment asks about hypothetical MIPS instruction `addsc` (scaled addition) that was the subject of 2014 Homework 3 Problem 3. See that assignment and its solution for a description of the `addsc` instruction.

Problem 1: Appearing below is a solution to 2014 Homework 3 Problem 3, though not the same as the posted solutions. Three of the multiplexors have labels on their select signals: A, B, and C.



The incomplete pipeline execution diagram below shows the progress of instructions through the implementation and also the value of the select signals A, B, and C in some cycles. If a select signal value is blank, such as C in cycle 5, then its value does not matter. For example, execution would be correct whether $C = 0$ or $C = 1$ in cycle 5, and so it is blank.

☑ Fill in instructions, including at least one `addsc`, that could have resulted in the execution. ☑ Take care to choose registers so that dependencies and ☑ the use of bypass paths are consistent with the select signal values.

Solution on next page.

The solution appears below.

Signal A selects a value headed for two possible destinations, the Left Shift unit in EX and the D In connection to Mem Port in the ME stage. The Left Shift unit in EX is only used by `addsc` and the D In connection is only used by store instructions (`sw`, for example). There for if a value is shown for A the instruction in EX must be either an `addsc` or some kind of store instruction. The value of A indicates whether the RT value is from the register file, $A = 1$, bypassed from ME, $A = 0$, or bypassed from WB, $A = 2$.

Signal B selects a value for the adder used by `addsc`; if $B = 0$ the value is from ME.ALU (which means it is probably not bypassed), if $B = 1$ the value is bypassed from WB. If a value is given for B then the instruction in ME must be an `addsc`.

If signal $C = 1$ the value to be written back (in the next cycle) comes from an `addsc` instruction, if $C = 0$ the output of ME.ALU is used. Signal C is blank for load instructions, and for instructions that don't write back at all.

For the first instruction A is blank (in cycle 2 when it is in EX) so it can't be an `addsc` nor a store. When the first instruction is in ME B is also blank, which is consistent with A being blank in the previous cycle. But $C = 0$, telling is that the instruction writes a result coming from the ALU. Any arithmetic or logical function would do, a `sub` was chosen.

For the second instruction $A = 0$ when it is in EX, indicating that it is an `addsc` or store, and that it is bypassing the result of the previous instruction. Because it is bypassing a value the `rt` register of the second instruction and the destination of the first must be the same. Register `r3` was chosen. Then the second instruction is in ME the $B = 1$ and $C = 1$ values tell us that it is an `addsc` and that the `rs` value is bypassed from the preceding instruction, so the `rs` register for the `addsc` is also `r3`.

The reasoning for choosing the next two instructions is similar.

```
# Cycle                0  1  2  3  4  5  6  7
A                               0  2  2
B                                  1     0
C                               0  1     1
# Cycle                0  1  2  3  4  5  6  7
sub R3, r5, r6         IF ID EX ME WB
addsc R1, R3, R3, 4       IF ID EX ME WB
sw R3, 0(r8)                 IF ID EX ME WB
addsc r5, r9, R1, 9             IF ID EX ME WB
# Cycle                0  1  2  3  4  5  6  7
```

Problem 2: Consider the *load/use* stall in the execution of the code below on an ordinary MIPS implementation (one without addsc):

```
# Cycle              0  1  2  3  4  5  6
lw r2, 0(r4)         IF ID EX ME WB
add r1, r2, r3          IF ID -> EX ME WB
```

(*a*) Suppose that instead of the code above the assembly code were generated by a compiler that **is aware** of the addsc instruction and run on an implementation that implements addsc.

☑ Explain how the compiler could avoid the stall.

It is the compiler that reads source code in some high-level language and emits assembly language instructions based on the source code. Of course, the compiler could avoid the stall in the usual way by separating the lw and add with some useful instruction. However, the question is about how a compiler aware of addsc could eliminate the stall.

The add is stalling because the load value arrives in ME near the end of the clock period in cycle 3, and so there is no way to bypass it to EX where the add needs it. But, an addsc instruction does not need its rt value until it is in the ME stage, and so it could bypass it. Knowing this, the compiler could emit an addsc r1, r2, r3, 0 instead of the regular add instruction. That's shown below.

```
# Cycle              0  1  2  3  4  5  6
lw r2, 0(r4)         IF ID EX ME WB
addsc r1, r2, r3, 0     IF ID EX ME WB
xor   r4, r5, r6           IF ID EX ME WB  <- Trouble if r5 changed to r1
```

Using addsc is fine in the code above with the xor. But if one of the sources of the xor were r1 then the xor would have to stall, meaning that substituting and addsc for an add this way eliminates one stall but adds another, and so there is no net gain. Having to consider all of these possibilities is why people who write compiler optimization code deserve great respect.

(*b*) Suppose instead that the original code (at the beginning of the problem) is run on an implementation which includes addsc and where addsc was encoded (choice of opcode, register fields, etc.) to avoid such stalls. (This could be the same implementation as the previous part.)

☑ Explain how such a stall could be avoided on the original code, with the add, by the design of the encoding of addsc.

In this problem the original code must be used as-is, with the add instruction. Suppose in the design of addsc the opcode and func field value for the addsc instruction were the same as those of the add instruction: opcode 0 (type R) and func field value 20_{16}. For the add instruction the sa field is defined to be zero. For addsc the sa is the shift amount. That means the encoding of addsc r1, r2, r3, 0 is identical to add r1, r3, r3. The hardware might execute add instructions the same way it executes addsc, meaning it would use the ME-stage adder. If so, then the stall above is avoided. It might also try to be smart about it, treating an add like an addsc only if that avoids a stall. *Possible midterm question?*

There's another problem on the next page.

Problem 3: Design the following control logic. Some of the logic will need the isADDSC logic block in ID, which detects whether an `addsc` instruction is in ID. An SVG of the diagram can be found at `https://www.ece.lsu.edu/ee4720/2025/hw03-scadd.svg`. It can be edited by Inkscape or any other SVG editor, and by plain-text editors for those who are so disposed.

- [x] Design control logic for select signal C. *Note: This is easy.*
- [x] Design control logic for select signal B.
- [x] Show control logic generating a stall signal for the stalls like those shown in the diagram below.

```
# Cycle                0  1  2  3  4  5  6
addsc r1, r2, r3, 4    IF ID EX ME WB
add r4, r1, r5            IF ID -> EX ME WB


# Cycle                0  1  2  3  4  5  6
lw r3, 0(r4)           IF ID EX ME WB
addsc r1, r2, r3, 4       IF ID -> EX ME WB
```

Solution appears below. Notice that the control signals are computed in `ID` and then carried through the pipeline to `ME` where they are used. Remember that when `addsc` is in `ID` some other instruction is in `ME`.

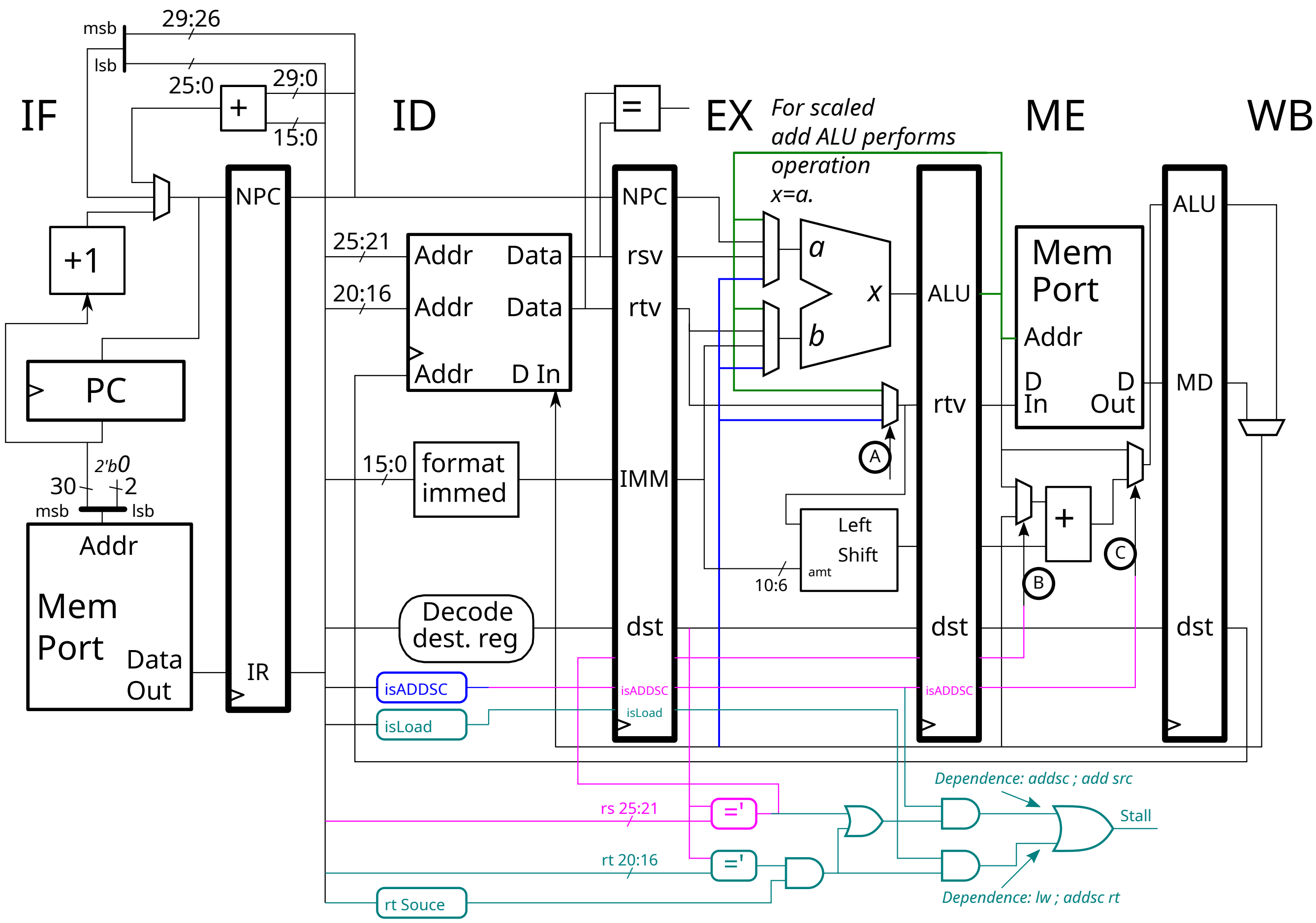
It should be easy to see that C should be 1 iff there is an `addsc` in `ME`, so compute the value in `ID`, and carry it along the pipeline until it is needed in `ME`. Signal B should be 1 when there is a dependency with the prior instruction. That is computed in `ID` by the purple comparison unit and then carried along the pipeline. Because B is only used for `addsc` instructions one might be tempted to put an AND gate in there to check. But there's no need to do so because it doesn't matter what value B is when the instruction is *not* an `addsc`, so there's no point wasting an AND gate.

The stall signals are computed by checking dependencies. For the first code fragment the control logic generates the stall in cycle 2 (the arrow head is where the stall ends, it starts in `ID`) when the `add` is in `ID` and the `addsc` is in `EX`. The logic compares the destination register of the instruction in `EX` (`r1` for the fragment) against the `rs` and `rt` sources of the instruction in `ID`. The logic assumes that the instruction uses `rs` as a source (not wise) but uses the rt Source logic block to check whether the instruction uses `rt` as a source. (For example, `add r1, r2, r3` uses `rt`, register `r3`, as a source but `addi r1, r2, 4` does not use `rt` as a source. [The `rt` field holds the destination, `r1`, in this instruction.]) The stall signal for the first code example is labeled *Dependence: addsc ; add src* in the diagram.

For the second code fragment the logic checks just for an `rt` dependence, because there would be no need to stall if the dependence were through the `rs` register. That is if the `lw` wrote `r2` instead of `r3` there would be no need to stall.

The control logic for A was not part of this problem. Designing that logic was asked on Problem 3c in the Fall 2003 Final Exam.

Diagram on next page.



IF
ID
EX
ME
WB
For scaled add ALU performs operation x=a.
msb
lsb
29:26
25:0
29:0
15:0
+
=
+1
NPC
IR
PC
Addr
Mem Port
Data Out
30
2'b0
2
25:21
20:16
Addr
Data
D In
15:0
format immed
Decode dest. reg
isADDSC
isLoad
rt Souce
rsv
rtv
IMM
dst
a
b
x
ALU
Left Shift
amt
10:6
A
B
C
Addr
D In
D Out
MD
rs 25:21
rt 20:16
Dependence: addsc ; add src
Dependence: lw ; addsc rt
Stall