CS 21 Project 2: Arduino-based Pipelined Processor

Department of Computer Science
College of Engineering
University of the Philippines - Diliman

1 Project Specifications

For this project, you are to create a simulation of the optimized five-stage pipelined MIPS processor using an Arduino.

1.1 Server program

Your Arduino implementation should be able to communicate with a server program via the Serial interface. The server program acts as the `PC` register, instruction memory, register file, and data memory components. Inputs to these components should be sent by the Arduino for every clock tick with the corresponding outputs sent by the server to the Arduino immediately as a response.

The server contains the actual MIPS program instructions to be executed. Instruction fetches must be done in a manner exactly how the standard five-stage pipelined MIPS processor would (details below).

The code of the Python-based server program will be available in UVLê. The program depends on the `PySerial` Python package to be installed (type `pip install pyserial` in your terminal to install this). You may ask for assistance regarding this.

1.2 Pipeline stages

The five standard MIPS pipeline stages are to be supported (included are relevant components after introducing hazard-resolving optimizations):

1. Instruction fetch (IF)
    - PC
    - Instruction memory
2. Instruction decode (ID)
    - Register file (read)
    - Branch resolution
3. Execute (EX)
    - ALU
    - Forwarding unit
4. Memory access (MEM)

    - Data memory

5. Write-back (WB)

    - Register file (write; negative edge-triggered)

1.3 Supported instructions

Only the following MIPS instructions (11) need to be supported:

- Pseudoinstructions (1)
    - `nop`
        - `0000 0000 0000 0000 0000 0000 0000 0000`
        - `do nothing`
- R-type instructions (5)
    - `add`
        - `0000 00ss ssst tttt dddd d000 0010 0000`
        - `$d = $s + $t`
        - Throws an `arithmetic overflow exception` on overflow
    - `sub`
        - `0000 00ss ssst tttt dddd d000 0010 0010`
        - `$d = $s - $t`
        - Throws an `arithmetic overflow exception` on overflow
    - `and`
        - `0000 00ss ssst tttt dddd d000 0010 0100`
        - `$d = $s & $t`
    - `or`
        - `0000 00ss ssst tttt dddd d000 0010 0101`
        - `$d = $s | $t`
    - `slt`
        - `0000 00ss ssst tttt dddd d000 0010 1010`
        - `if $s < $t, $d = 1; else $d = 0`
- I-type instructions (4)
    - `addi`
        - `0010 00ss ssst tttt iiii iiii iiii iiii`
        - `$t = $s + imm`
        - `imm` is the 16-bit immediate sign-extended to 32 bits interpreted as 2C (applies to all below)
        - Throws an `arithmetic overflow exception` on overflow
    - `lw`
        - `1000 11ss ssst tttt iiii iiii iiii iiii`
        - `$t = MEM[$s + imm]`
        - Assume word alignment is always followed
    - `sw`

        - `1010 11ss ssst tttt iiii iiii iiii iiii`
        - `MEM[$s + imm] = $t`
        - Assume word alignment is always followed
    - `beq`
        - `0001 00ss ssst tttt iiii iiii iiii iiii`
        - `if $s == $t, PC += (4 + (imm << 2)); else PC += 4`

- J-type instructions (1)
    - `j`
        - `0000 10nn nnnn nnnn nnnn nnnn nnnn nnnn`
        - `PC = (PC & 0xf0000000) | (N << 2)`

1.4 Exceptions

- Invalid/reserved instruction exception
    - All machine instruction bit combinations not supported (see above) must raise an exception after being decoded in the ID stage
    - The exception handler address for this exception is `0x80010000`
- Arithmetic overflow exception
    - If the result of `add`, `sub`, or `addi` causes an overflow (i.e., `(+) + (+) = (-)` or `(-) + (-) = (+)` after applying signs), an arithmetic overflow exception must be raised
    - The exception handler address for this exception is `0x80020000`

1.5 Hazards

No structural hazards are assumed to exist. Only the hazards below are to be detected (included are the resolutions to be implemented).

1.5.1 Control hazards

- The branch prediction scheme is `Predict not taken`
- Incorrectly-fetched instructions in the pipeline must be flushed (i.e., must be converted into `nop`s) when a `beq` or `j` instruction gets to the EX stage
- Instructions fetched *before* the flush-triggering `beq` or `j` are unaffected
- Note that there are no assumed delay slots for `beq` and `j`

1.5.2 Data hazards

- EX-detected forwarding-resolved hazards
    - These hazards are detected when the later instruction of the pair is in the EX stage
    - Assuming A and B are the data hazard instruction pair with B coming after A, the destination of A must be equal to a source of B
        - `RD` as the destination:

            - `add`, `sub`, `and`, `or`, `slt`
        - `RT` as the destination:
            - `addi`, `lw`
        - `RS` as a data source:
            - `add`, `sub`, `and`, `or`, `slt`
            - `addi`, `lw`
        - `RT` as a data source:
            - `add`, `sub`, `and`, `or`, `slt`
    - When overlapping data hazards coexist in the pipeline (i.e., both **Case 1** and **Case 2** are present), the forwarding of **Case 1** data is prioritized
        - **Case 1:** Instructions of hazard pair are next to each other
        - **Case 2:** Instructions of hazard pair have one instruction between them

- ID-detected forwarding-resolved hazards
    - This class of hazards is specific to `beq` as its resolution is performed during the ID stage
    - Data needed for branch resolution is potentially forwarded from later stages
    - Assuming A and B are the data hazard instruction pair with B coming after A, the destination of A must be equal to a source of B
    - When overlapping data hazards coexist in the pipeline (i.e., both **Case 1** and **Case 2** are present), the forwarding of **Case 1** data is prioritized
        - **Case 1:** Earlier instruction is in the EX stage
        - **Case 2:** Earlier instruction is in the MEM stage
    - Note that it is not a hazard if the earlier instruction of the pair is in the WB stage since the data will be committed to the register file and the branch resolution can use the updated data in the register file with no forwarding needed

- Load-after-use
    - Load-after-use hazards happen when there is a data memory load followed by use of loaded data (e.g., `lw` to `$t3` followed by `add` using `$t3`)
    - These are detected when the instruction using the loaded data (e.g., `add`) has reached the ID stage (with the `lw` instruction in either EX or MEM)
    - When a load-after-use hazard is detected, a `nop` must be inserted in the EX stage (right after the triggering instruction) during the next clock cycle while retaining whatever instructions are in IF and ID (i.e., the first two stages are frozen for one clock cycle) as a way of *stalling*

1.6 Input/output components

The circuit must have the following components on a breadboard:

- Tact switch (`Start`)
    - This button must be pressed to (re)start testing
    - Pressing this button must trigger sending a `START` message to the server; another `START` message will be sent by the server as a reply

- Tact switch (`CLK`)
  - Pressing this should trigger a single-stage movement of the pipelined processor instructions; the Arduino must send data input messages and expect data output messages as a response (see format below)
- Two (2) 7-segment LEDs showing the last two decimal symbols of (`PC >> 2`)
- LEDs (5 in total)
  - Ready
    - This should be activated when the Arduino receives a `START` message from the server
    - This should be deactivated when the `CLK` button is pressed
  - Data hazard
    - This should be activated while the instruction in the EX stage has a data hazard with either the MEM instruction, WB instruction, or both
  - Control hazard
    - This should be activated while the branch/jump instruction in the EX stage results in a branch/jump (i.e., incorrect instructions have been loaded into the pipeline)
  - Invalid/reserved instruction exception
    - This should be activated when an invalid/reserved instruction exception has been raised
    - This should be deactivated when `Start` is pressed; it must stay activated until the `Start` button is pressed
  - Arithmetic overflow exception
    - This should be activated when an arithmetic overflow exception has been raised; it must stay activated until the `Start` button is pressed

1.7 Serial input/output

A mode of communication exists between the Arduino and the server implemented with details below. **The project submission is invalid if this is not satisfied**.

1.7.1 Data input messages

The messages below must be sent by the Arduino to the server via the Serial interface (serial output).

- `START` initiation message
  - The server sends a message containing `START` terminated by a null character
  - This message is sent when the `Start` tact switch is pressed
  - All server-side data gets reinitialized upon receipt
- `INPUT` message
  - The server sends a message with the format `PPPPPPPP AAAAA BBBBB CCCCC DDDDDDDD E FFFFFFFF GGGGGGGG H I` (null-terminated, single space-delimited) as described below:

    * `PPPPPPPP` is a 32-bit value in hex (lowercase, no prefix, all leading zeros shown) corresponding to the input into the `PC` register
    * `AAAAAAAA` is a 5-bit value (`00000`-`11111`) corresponding to the `Read Register 1` input of the register file
    * `BBBBBBBB` is a 5-bit value (`00000`-`11111`) corresponding to the `Read Register 2` input of the register file
    * `CCCCCCCC` is a 5-bit value (`00000`-`11111`) corresponding to the `Write Register` input of the register file
    * `DDDDDDDD` is a 32-bit value in hex (lowercase, no prefix, all leading zeros shown) corresponding to the `Write Data` input of the register file
    * `E` is a 1-bit value (`0` or `1`) corresponding to the `Write Data` input of the register file
    * `FFFFFFFF` is a 32-bit value in hex (lowercase, no prefix, all leading zeros shown) corresponding to the `Address` input of the data memory
    * `GGGGGGGG` is a 32-bit value in hex (lowercase, no prefix, all leading zeros shown) corresponding to the `Write Data` input of the data memory
    * `H` is a 1-bit value (`0` or `1`) corresponding to the `MemRead` input of the data memory
    * `I` is a 1-bit value (`0` or `1`) corresponding to the `MemWrite` input of the data memory
  - The data in the message modifies the internal state of the said components and will affect their output values as seen in the `OUTPUT` message

1.7.2 Data output messages

The messages below will be sent by the server to the Arduino via the Serial interface (serial input) as responses to the data input messages above.

- `START` confirmation message
  - The server sends a message containing `PPPPPPPP` in hex (lowercase, no prefix, null-terminated, all leading zeros shown) corresponding to the initial value of `PC` to be used for fetching the first instruction after the next clock tick
  - This message is sent as a response to the `START` message sent by the Arduino
  - This value (divided by 4) should be displayed in the 7-segment LEDs immediately
  - Assume `PC` is always properly word-aligned
- `OUTPUT` message
  - The server sends a message with the format `IIIIIIII AAAAAAAA BBBBBBBB CCCCCCCC` (null-terminated, single space-delimited) as described below:
    * `IIIIIIII` is a 32-bit value in hex (lowercase, no prefix, all leading zeros shown) corresponding to the `Instruction` output of the instruction memory
    * `AAAAAAAA` is a 32-bit value in hex (lowercase, no prefix, all leading zeros shown) corresponding to the `Read Data 1` output of the register file
    * `BBBBBBBB` is a 32-bit value in hex (lowercase, no prefix, all leading zeros shown) corresponding to the `Read Data 2` output of the register file
    * `CCCCCCCC` is a 32-bit value in hex (lowercase, no prefix, all leading zeros shown) corresponding to the `Read Data` output of the data memory

- This message is sent as a response to the `INPUT` message sent by the Arduino
- The data in the message indicates the outputs of the said components and reflects their internal state as modified by the sent `INPUT` messages so far

1.8 Notes

- Assume that the pipeline stages all contain `nop` instructions initially (i.e., when `Start` is pressed)
- As there is no support for the `syscall` instruction, assume that programs do not terminate
- Programs may have a self-referential branch/jump instruction to simulate termination

2 Grading

2.1 Scoring rubric

Criterion	**Points**
All tact switches and non-hazard LEDs work as intended	10
All 7-segment LEDs work as intended	10
`add` works as intended in isolation	5
`sub` works as intended in isolation	5
`and` works as intended in isolation	5
`or` works as intended in isolation	5
`slt` works as intended in isolation	5
`addi` works as intended in isolation	5
`lw` works as intended in isolation	5
`sw` works as intended in isolation	5
`beq` works as intended in isolation	5
`j` works as intended in isolation	5
Invalid/reserved instruction exceptions are properly handled	15
Arithmetic overflow exceptions are properly handled	15
Data hazards are detected and properly handled	30
Control hazards are detected and properly handled	10
	140/100

2.2 Project checking

Grading of the project will be done both via automated and manual testing.

3 Submission Details

Submit your Arduino code with the filename `cs21172project2.ino` through the UVLê submission module.

At least one group member must be present during project checking.

Soft deadline (INC): 2018 June 1, 5:00pm
Hard deadline: 2018 July 26, 5:00pm

Note that project submission and checking may be done **earlier** than the specified dates below in case of scheduling concerns.