Project #3 Pipelined MIPS

Due date: May 30th

Seehwan Yoo

Dept. of Mobile Systems Engineering

Dankook University

Analogy with previous project

- Make MIPS CPU emulator
- Execute MIPS instructions (except for floating point ops)
  - All of the front page instructions in Green sheet
- Assume that you have memory, and program is loaded in memory before execution
- Take MIPS binary program
  - You can generate it with mips-gcc
- Display the architectural state changes at every clock cycle
- Handle exception gracefully

DANKOOK UNIVERSITY

Introduction to pipeline

- Execution is performed through multiple stages
  - IF (instruction fetch)
  - ID (instruction decode)
  - EXE (instruction execution)
  - MEM (memory access)
  - WB (write back result)
- We assume each stage takes one cycle
  - i.e. 5 cycles per instruction execution
- We overlap execution of multiple instructions
  - Ideally 5 instructions run at the same time
    - Max. 5 in-flight instructions in pipeline

Simulate pipeline in software

- Cycle is updated at the end of instruction execution loop
  - Until the cycle is changed, all hw operations are assumed to be executed in the same cycle
- There are multiple in-flight instructions
  - Use latch to hand over the stages
  - Strictly depend upon the previous stage's result latch
  - Produce output on output latch
    - Do NOT access global things
  - Use input on input latch
- At the end of cycle, latch handover is made
  - Output latch values flow into input latch (in the next stage)

An example) IF->ID (cont')

- Basic knowledge
  - IF stage: read memory in which PC points to
  - ID stage: decode instruction, and read register values
- At the beginning:
  - IF reads memory, and put the read value (instruction) on IF/ID latch
- At the same time:
  - ID gets the instruction from IF/ID latch,
  - Decode instruction (opcode, rs, rt, rd, control signals),
  - Read registers for rs, rt (if necessary),
  - Perform sign-extension for imm,
  - and put the produced value on ID/EX latch
    - Read register values, sign-ext-imm, etc.

An example) IF->ID

- At the same time:
  - ID gets the instruction from IF/ID latch,
  - Decode instruction (opcode, rs, rt, rd, control signals),
  - Read registers for rs, rt (if necessary),
  - Perform sign-extension for imm
  - and put the produced value on ID/EX latch
    - Read register values, sign-ext-imm, etc.
- At the end of cycle:
  - IF/ID latch is filled by IF
  - ID/EX latch is filled by ID

Wrong way!

- Note that multiple instructions are in-flight
  - IF instruction, ID, EX, MEM, WB has all different instructions
- Bad way to go
  - Inside execution loop,
  - IF produces result on IF/ID latch
  - ID should take input from IF/ID latch

```
while (1) {
  if_latch = fetch_inst();
  ex_latch = decode_inst(if_latch);
  // if_latch has overwritten by fetch_inst()!
  …
}
```

Another way (use input/output latches)

- ID runs with input latch
  - Which holds instruction
- EX runs with input latch
  - Which holds decoded instruction and read register values
- …
- ID produces result on output latch
  - Which holds decoded instruction and read register values
- EX produces result on output latch
  - Which holds on ALU calculation result, memory address
- At the end of cycle,
  - Move output latch values → next stage input latch

Data Dependency handling

- Pipeline with data dependency
  - Detect and wait
  - Detect and forward/bypass
  - Detect and eliminate
  - Predict and verify
  - Do something else
- My suggestion (or additional implementation)
  - Stalling
  - Forwarding
  - Scoreboarding

Control dependency handling

- Pipeline with control dependency
  - Stalling
  - Branch prediction
  - Branch delay slot
  - Fine-grained multi-threading
  - Predicated execution
  - Multipath execution
- My suggestion (or additional implementation)
  - Stalling
  - Branch delay slot
  - Branch prediction

NOTE:

- This program assignment is critical in your evaluation, so please work hard to complete in your schedule. If you need help, please ask for the help. (I am here for that specific purpose) I, of course, welcome any questions on the subject.

- We will have demo time for some of your work (good/bad). In demo, you are asked to explain your software (structure/implementation). If you can, please think about the visualization of the emulator.

- Note for the one strict rule that do not copy code from any others. Deep discussion on the subject is okay (and encouraged), but same code (or semantics) will result in sad ending.

Requirements

- The Objective: compare performance of pipelined vs. single-cycle u-processor.
  - Your program should produce correct output. (and execution should be the same with code semantics)
  - Compare the number of CPU clock cycles for single-cycle execution and pipelined execution. The clock cycles should be reasonably increased.
  - In theory, we can reduce CPU clock cycle time by 1/5 in 5-stage pipelined execution.

Machine Initialization

- Before the execution, the binary file is loaded into the memory. Note that memory can be a data structure defined with large array.
- Read all the file content into your memory (data structure).
- Assume that initial RA value is 0xFFFF:FFFF.
  - Thus, when your PC becomes 0xFFFF:FFFF, your machine completes execution, and halts.
- Your application is loaded to 0x0, and initial stack pointer is 0x100000.
- Rest of registers are initialized with value zero.

Implementation requirements:

- You should implement five-stage pipelined MIPS processor emulator. The emulator should implement IF, ID, EX, MEM, WB stages.
  - For each stage, instruction execution has to be latched. That is, you have to store the execution state for each stages. Latches are the temporal storage that remember the execution states for each stages execution.
- You can have separated instruction memory and data memory.
- The emulated processor runs with the emulated clock cycle.
  - You need to generate clock, which is a variable. The clock cycle increases only after all the work done in the all the stages. Note that max. 5 different instructions can run at the same cpu clock cycle.
- You need to resolve data dependency among instructions. That is, you have to implement either stall, forwarding or register renaming. Forwarding is recommended as a basic implementation, but you can optionally choose to implement stall. You should make sure that it works, as in the program order (semantics).
- You need to resolve control dependency among instructions. According to the MIPS ISA, one delayed branch slot is defined. Invalidation is basic implementation, but you can optionally implement branch prediction mechanisms.

Output

- Output:
  At the end of each cycle, the simulator prints out
  - the changed micro-architectural state from the previous state.
  - Micro-architectural state consists of set of general register, PC, and memory, and latches, internal data structure.
  - You can print out only changed state.

- After the execution completion, the simulator prints out
  - Final result (r2)
  - The total number of execution cycles
  - Instructions execution statistics
  - Total # instructions
  - # of memory ops
  - # of reg. ops
  - # of (conditional) branches
  - # of not-taken branches
  - # of jumps

- Think about good presentation of pipelined execution.
- You can give log option in build time (e.g. #define DEBUG_IF)

Evaluation:

- Different credits are given for implementations, and demos.
- More credits are allotted for different (hardware-based) hazard resolving techniques such as branch prediction, forwarding, stalling, etc.

Input program binary:

- You can use your own MIPS code (compiled) or download binary.
- To make MIPS binary, use MIPS cross compiler toolchain.
  - First, write C code, and compile mips-linux-gnu-gcc with "-c -mips1" option (compile only).
  - Second, translate the object binary, stripping ELF headers. mips-linux-gnu-objcopy –O binary –j .text input.o input.bin.
  - Third, check the integrity between binary files: objectdump, mips-linux-gnu-objdump –d input.o , vi –b input.bin → check with :%!xxd option. (google internet) or hexedit tool
- To use with multiple functions, you need to hand-carve binary for function calls (jal 0, originally). For example, jal to 0x40 can be encoded as (0x0C000010).
- Sample input file examples can be downloaded from the e-learning site. Two representative example programs are
  - 1) summation from 1 to 10,
  - 2) return 10,
  - 3) calculating 4 fibonacci,
  - 4)101-th smallest number from 10,000 random numbers
- Some additional good examples are capturing N-way of counting coins, hanoii towers movements. Students who took system programming course can try them to implement in C/C++ language.

Some advices

- Advice one: You may need some time to think about your software structure, and operation, and pipeline. Therefore, think with note and pen, before rushing the code work.

- Advice two: Make basic structure robust, and reliable. If you make code on suspicious base, you are easy to lose the way. Make print logs, before you have gone too much. Make basic structure, at the first hand. Make some checkpoints before you got failure.

Strategic movements

- Advice three: You can make things work from small pieces to a larger one. You may consider writing code in the following sequences.
  - Make sure you have sound single cycle implementation
  - Establish pipeline structure
  - Make debug function
  - Make basic control flow work (sequential execution path)
  - Make reg. memory access instructions work
  - Make pipeline with forwarding
  - Make jump/branch work
  - Make branch prediction

Lastly,

- Last advice, but not least: Begin as early as possible. Ask for help as early as possible. Try as early as possible. They are for your health-care.

- Good Luck!