NANYANG TECHNOLOGICAL UNIVERSITY

CZ3001 Advanced Computer Architecture
AY 2013-14

Interim Report

Chaudhuri Kaustav U1121685K
Chikersal Prerna U1120672G
Ong Wei Siang U1122334J

1. Introduction

In this project, we implement a simple 16-bit CPU with a load/store architecture, using the ALU and Register File implemented in Lab 1 and 2, as well as a modified version of the control/ data-path implemented in lab 3.

This CPU consists of the following major components:
(1) the arithmetic logic unit, which performs calculation and computation of instructions,
(2) a register file, the main storage location for application data while the CPU is working on it,
(3) a simple Control unit/Data-path of a processor.

It also contains a 3-bit FLAG register (Zero (Z), Overflow (V), and Sign bit (N)) and 16 instructions, which can be classified into the following types:

- Arithmetic Instruction
- Memory Instruction
- Load immediate instruction
- Control instruction

Furthermore, the project is split into 2 phases. An overview of these phases can be found in sections 2 and 3.

2. Phase 1

Phase 1 involves implementation of a five-stage pipeline (IF-ID-EX-MEM-WB), and hazard detection to ensure that all the data and control dependencies are handled. The data memory and instruction memory are kept separate in this phase. This section describes pipelining and the stages we implemented.

2.1. Pipelining and its advantages

Pipelining helps improve the *throughput* of the CPU. The five stages of pipelining are separated by special registers called pipeline registers. The purpose of these registers is to separate the stages of the instructions, such that data conflicts will not occur due to execution of multiple instructions at the same time. Having more pipeline stages increases the number of instructions that can be executed per second. It also allows us to use the hardware more efficiently, since none of the stage needs to include a complete hardware control unit. Each stage only needs its own dedicated hardware to perform its specific task and is able to run in parallel with the other stages.

2.2. Pipelining, its disadvantages and hazards

Pipelining does have certain disadvantages, such as, it increases the latency of individual instructions because of the added pipeline registers. Also, there is an increased area and power cost for the pipeline registers. These issues are not very significant as we are mainly concerned about increasing the throughput of the CPU in order to decrease the time taken by a computer program to perform a task. Pipelining helps us achieve this, however, one big issue with pipelining is that it is only effective if you have proper instructions to feed the pipeline. For example, dependencies can be a problem with a deeply pipelined CPU. Unless dependencies can be handled through bypassing, stalls/pipeline bubbles may be introduced which decrease the throughput achieved.

Situations which prevent execution of the next instruction during its designated clock cycle are known as *hazards*. Hazards are of 3 kinds:

- *Structural Hazards*: They arise from resource conflicts when the hardware cannot support all possible combinations of instructions in simultaneous overlapped execution.
- *Data Hazards*: They arise when an instruction depends on the result of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline.
- *Control Hazards*: They arise from the pipelining of branches and other instructions that change the PC.

Hazards can be resolved by stalling the pipeline, that is, eliminating a hazard often requires that some instructions in the pipeline to be allowed to proceed while others are delayed. When the instruction is stalled, all the instructions issued later than the stalled instruction are also stalled. Instructions issued earlier than the stalled instruction must continue, since otherwise the hazard will never clear.

2.3. Our five-stage pipeline

The five stages of the pipeline in our design in as follows:

(1) Instruction Fetch *(IF)* Stage:

The first stage is to fetch the instruction. Instruction is loaded from memory. The address of the instruction in memory is decided by the hardware, according to the control signals received from other stages. The instruction would be written into the IF/ID register at the next positive edge of the clock cycle. The writing of the instruction tells the Instruction Decode (ID) Stage to begin its operations on the instruction.

The IF stage includes the program counter (PC), instruction memory (IMemory: instantiation of "memory" module), addPC and IF_ID modules.

The *PC* has a 16-bit input, 1-bit clock, 1-bit reset and a 16-bit output. After the reset, value of PC is set to 0. It is being used as the first address from the first instruction read from the text file through *IMemory (instantiation of "memory" module)*. This address is used by the instruction memory to fetch the instruction corresponding to the address. The *addPC* module will increment the PC value by 1 so the fetching module can use it to fetch the next instruction. The *IF_ID* module passes the instruction and its PC value to the decoding (ID) stage to decode the instruction. The cycle repeats again as the next PC value is being output into the addPC module to be incremented after the instruction for it has been fetched. This stage does not need the hardware used to decode or execute instructions.

(2) Instruction Decode *(ID)* Stage:

This stage would not need to access the memory because the previous IF stage has already loaded the instruction from memory into one instruction register. Instead, this stage would decode the instruction. It simply reads the register file, and generates data-path decode/control signals.

Control signals are determined using the first 4 bits of the instruction, called the Opcode. The signals are divided into their corresponding stages which they control and are passed along through the pipeline registers together with the other data. The instruction also contains the address of the registers to be used.

As for the load instruction, the memory will first be read, data written back to register from memory. It would then enable the write signal of the register file, and also take the ALU calculation output of offset. For the store instruction, data would not be written to memory. All the load immediate instructions (for example LLB and LHB) and ALU operations also allow data write and register file write.

(3) Execution *(EX)* Stage:

Now we come to the execution stage. This stage would get data from the ID/EX register. If the instruction is an arithmetic or logical operation, the result is computed. If it is a load-store instruction, the address is computed. This is performed by an electronic circuit called the ALU which performs all the calculations, for instance, ADD and SUB. The result of the ALU operation is stored into EX/MEM register along with the control for the MEM and WB stages forwarded from the ID/EX Register.

(4) Memory *(MEM)* Stage:

The memory stage's load or store data from or to memory address calculated in EX stage. The control signals passed in from the EX/MEM register determine which of the operations to perform. The output of

the memory is written into the MEM/WB register along with the WB control that is passed from the EX/MEM register.

(5) Write Back *(WB)* Stage:
The write back stage writes the result into the register file of the EX/MEM register or from ALU output and stores the resulting value from the operation back to the register file. The Multiplexer in WB stage chooses the data from the EX stage (for arithmetic/logical instructions) or MEM stage (for LOAD instructions), to be written to the destination register. The write back is connected to the data write port of the register file in the decode stage. As for JAL, this will contain register 15 that will be put into the write port, in order to store the PC value.

3. Phase 2
In phase 2, we will replace the individual instruction and data memories with a single shared memory which has a latency of 3 cycles for read and write. Because we share main memory, the instruction fetch and memory pipeline stage cannot access the main memory concurrently. So, we will need to implement selection logic to pick which one accesses the memory, as well as handle the structural hazard of sharing the memory.

The details of phase 2 will be covered in our final report.

4. Features implemented
Due to time constraint and team member related issues, such as those described in section 7 of this report, we could manage to implement only a part of phase 1. The features implemented are as described in section 4.1. We will be implementing the remaining portion of phase 1 and phase 2 by the demo day. Our further plans are described in section 6.

4.1. Phase 1
We have implemented the following features in Phase 1:
1. Instantiated the instruction memory and the data memory, which are kept separate in this phase.
1. The five-stage pipeline as described in section 2.3.
2. Modified the Control unit to produce signals which allow operations performed by arithmetic and memory instructions.
3. Modified ALU which updates the 3-bit FLAG register in the CPU, and allows us to perform 8 arithmetic and logical operations: ADD, SUB, AND, OR, SLL, SRL, SRA, and RL.
4. Added support for LW and SW Memory instructions.

5. Interim Test results
Due to time constraint and team member related issues, such as those described in section 7 of this report, we could not test the features we have implemented. The testing we did while coding was to simply print out certain values. We will implement test benches in order to test our CPU design and present the results in our final report.

6. Further plans
We plan to implemented the following features in the coming weeks till the demo:
1. Add support for load immediate instruction (Phase 1)
2. Add support for control instructions (Phase 1)
3. Hazard detection to ensure that the pipeline correctly handles all data and control dependencies (Phase). The importance of hazard detection was described earlier in section 2.2.
4. Replace the individual instruction and data memory with a single shared memory (Phase 2).
5. Implement selection logic to decide how the single shared memory is accessed (Phase 2).
6. Implement logic to stall the pipeline to handle the structural hazard of sharing the memory (Phase 2).

7. Note on team contribution

Our team had a fourth member, named Runsheng Liu. She did not contribute to the project AT ALL, leaving us short of one person. Therefore, we have mutually agreed that she will not be a part of our team. ***We request you to revoke her access/pushing rights to our stash repository, as soon as possible, in order to prevent the use of unfair means.***

Among the three of us, while we did use multiple accounts to commit to stash, our commits would not be a fair way of judging our contributions, since we worked on the project sitting together, such that 2 people often worked on the same computer and hence committed from the same account.

Prerna and Kaustav contributed 40% EACH, while Wei Siang contributed to 20% of the project. However, in the coming weeks, we are sure that Wei Siang will contribute more. Overall, all three of us are happy to be working with each other and are making progress.

Individual file contributions are as follows:
(Please note: efforts required for implementing these modules are unequal)

- control.v: *Prerna and Kaustav*
- phase1_top.v: *Prerna and Kaustav*

- adder.v: *Kaustav*
- alu.v: *Kaustav*
- shift_left_2.v: *Kaustav*
- sign_ext_4_16.v: *Kaustav*

- andgate: *Prerna*
- ex_mem.v: *Prerna*
- id_ex: *Prerna*
- if_id: *Prerna*
- mem_wb.v: *Prerna*

- pc.v: *Wei Siang*
- addPC.v: *Wei Siang*
- MUX16.v: *Wei Siang*
- MUX4.v: *Wei Siang*

The interim report is written jointly by ALL THREE of us. The description of the implementation of the five-stages was written by Wei Siang alone.