Pipelined CPU

18-1 ECE 322 Computer Organization, Lab 06

Hansung Kim
2014-16824

1 Introduction

Understand why pipelining improves the performance of a CPU, what the control/data hazards are and how to resolve them, and design and implement a pipelined CPU.

2 Design

The biggest difference in design of a pipelined CPU from a multicycled one, is that the micro-controlled state logic implemented in the control unit is transformed into a datapath form that implements per-cycle register transfers. This is because multiple instructions from different time points share resources in a single cycle. This is made possible by physically segregating the resources into multiple stages, and delivering each control signals to the right stage, effectively "synchronizing" them with the instruction being executed. This segregation in turn is made possible by disallowing any resources to be reused by a single instruction. Thus, no resource recycling is possible for a pipelined CPU, such as ALU for both address calculation (ID) and integer operation (EX), as it was for the multicycle design. Instead, as multiple time slots effectively share the CPU resources available in a single cycle, the IPC is greatly improved compared to the multicycle design.

All major design solutions for implementing a working pipeline is referenced from P&H and the lecture materials, and only the design for pipeline hazard resolution is presented.

2.1 Data hazard resolution

Both stall and data forwarding method are implemented. The hazard unit module (`hazard_unit.v`) checks `reg_write` control signal and the destination register of the older instructions in EX, MEM and WB stage to determine whether the operand is bready to be decoded and fed into EX stage.

Data forwarding eliminates most of the stall by forwarding in-flight values in MEM or WB stage to the EX stage in the same cycle. As data forwarding requires different stall logic (LOAD-USE

stall), some stall condition check should be conditionally disabled in order to share the same code for two different hazard resolution methods. This is done by logical-ANDing the stall conditions with a configuration parameter (e.g. `!DATA_FORWARDING && [stall condition]`).

Register indirect jumps such as JPR and JRL require a special case of data forwarding, as they are resolved in the ID stage and not EX stage as it is for integer and other branch operations. Although it would be more ideal to add logic that forwards values to the ID stage (read register number) as well as the EX stage, this design does not implement such extended logic.

2.1.1 Pipeline stall for data hazards

Pipeline stall is implemented by (1) disabling PC and IR latching, and (2) "bubblifying" the instruction in ID stage by clearing all control signals to zero. This stalls the progress of program counter, and effectively turns the current instruction into a no-op. This is shown in the `Data hazard handling` section in `hazard_unit.v`.

2.1.2 Register file self-forwarding

In addition to ALU operand forwarding, the register file implements self forwarding of the WB value by writing the registers at the negative clock edge (`!clk`). This can be enabled or disabled by setting the module parameter.

2.2 Control hazard resolution

Both stall-based and branch prediction method are implemented. Branches are handled in different stages depending on the type: conditional branches are resolved in EX stage, whereas unconditional jumps are resolved in ID stage. While this is be better for decreasing the number of stalls and branch prediction miss penalty, this leads to some complexity in the design.

2.2.1 Pipeline stall for control hazards

Pipeline stall for data hazards and control hazards has a difference in that the former stalls and nullifies all stages containing wrongly fetched instructions, whereas the latter only stalls the ID stage. As such, the control hazard stall is a "flush" that completely invalidates all instructions younger than the branch resolution. However, "bubblifying" the control signals only help invalidating the stages from ID and on. Invalidating the IF stage is done by clearing the instruction register to a NOP instruction (predefined as `0xFFFF` in this design).

As conditional branches are handled in EX stage, it should flush both IF and ID stage, whereas unconditional jumps should only flush the IF stage. This is shown in the `Control hazard handling` section in `hazard_unit.v`.

2.2.2 Branch prediction

The following four branch prediction methods are implemented:

- Predict always untaken using PC + 1.
- Predict always taken using a BTB without BHT.
- Predict by 2-bit saturation counter using a BTB with BHT.
- Predict by 2-bit hysteresis counter predictor using a BTB with BHT.

Stall based resolution is different from all of this because it does not flush on prediction miss.

2.2.3 PC and num_inst update logic

Because the conditional branch and jump are handled in different pipeline stages, there may be cases where both a conditional branch (in EX) and a jump (in ID) are resolved in the same cycle. In these cases, the jump instruction in ID stage may or may not be a speculative instruction, depending on the resolution of the conditional branch in the EX stage. Thus, only after making sure that the branch in the EX stage did not miss can the jump instruction be assumed non-speculative handled thereafter.

In addition to this, if the conditional branch turns out as a miss and speculative instructions should be flushed, the `num_inst` should also be restored to the value it had before the first speculative instruction was fetched. This can also be done only when the branch in EX is made sure to be missed.

This together can be expressed as a pseudocode in Figure 1. The code is in "PC resolution" section in `datapath.v`.

```
if (cond_branch miss in EX)
   PC <= branch_target;
   num_inst <= num_inst_saved;
else if (uncond_branch miss in ID)
   PC <= jump_target;
else
   PC <= predicted nPC;
```

Figure 1: PC update and `num_inst` restore algorithm

The BHT update is done in a similar way, except that it should be updated regardless of the branch hit or miss, so it omits branch hit-or-miss check. The BTB tag update logic is simpler because the branch target (whether unconditional or jump) is resolved in the ID stage and the tag can be updated in the same stage.

3 Implementation

Measurements of the number of cycles taken to finish the given testbench for each case of pipeline hazard handling methods are presented here.

3.1 Comparison between pipeline and multicycle

The total number of instructions executed is shown as 981 in the `num_inst` variable. Dividing this by the number of clock cycles (`num_clock`) we can get the average IPC. As the clock time and number of instructions are unchanged from the multicycle testbench, we can directly compare the performance using the IPC.

Design	# of cycles	IPC
Multicycle	3536	0.2774
Pipelined (no data/RF forwarding, stall-on-branch)	2138	0.4588

We can see that the pipelined design gained overall performance speedup of 1.654 over the multi-cycle design.

The speedup is underwhelming considering the ideal IPC of the pipelined design is 1, disregarding any pipeline hazards. It is likely that our design suffered severe performance drop compared to the ideal case due to frequent pipeline stalls. The majority of these pipeline hazards are further eliminated using data forwarding and branch prediction whose performance are shown below.

3.2 Performance of data forwarding

Data forwarding can eliminate most of produce-use data hazards by fetching in-flight operand values from the future pipeline stages. Comparison of its performance to the stall-based approach is shown below.

Data hazard resolution	# of cycles	IPC
Stall until ready	2138	0.4588
Self-forwarding RF	1768	0.5549
Data forwarding with self-forwarding RF	1358	0.7224

Combined with RF self-forwarding, the forwarding method of resolving data hazards brought speedup of 1.575 over the stall-based method. The IPC gain of 0.2636 is 49% of the total IPC drop of 0.5412 from the ideal pipeline (1), indicating that the data hazard is the major source of pipeline hazards for this particular workload.

3.2.1 Self-forwarding RF

As shown in the design section, the self-forwarding of the register file (write at negative clock edge) is separately implemented and also compared no such improvement. Its performance is shown together in the above table. The RF self-forwarding alone gained speedup of 1.209 over the stall-based resolution.

Due to a bug in the implementation, the data forwarding implementation with RF self-forwarding disabled failed to finish and could not be compared.

3.3 Performance of branch prediction

Branch prediction is one way of resolving control hazards along with delay slots and compiler optimizations, and the only way implemented in this assignment.

There are four kinds of branch predictions implemented as shown in section 2.2.2, and the performance for each of them compared to the stall based implementation is shown below.

Branch predictor	# of cycles	IPC
Stall on branch	1358	0.7224
Always untaken, no BTB, flush-on-miss	1261	0.7780
Always taken, BTB without BHT	1141	0.8598
Saturation counter, 2-bit BHT	1173	0.8363
Hysteresis counter, 2-bit BHT	1165	0.8342

All measurements enable data forwarding and RF self-forwarding, and all four branch predictors use flush-on-miss policy. The accuracy of each predictor is also shown below:

Branch predictor	Hits	Misses	Total	Accuracy
Always untaken, no BTB, flush-on-miss	49	222	271	0.181
Always taken, BTB without BHT	172	99	271	0.635
Saturation counter, 2-bit BHT	156	115	271	0.576
Hysteresis counter, 2-bit BHT	160	111	271	0.590

The branch prediction actually performed worse on 2-bit BHT-based predictors than on the always taken predictor that only records branch targets on tag conflict, and does no BHT update. However this result is highly dependent on branch outcome patterns. For this particular workload, it seems that the outcome of `BGZ $1, FIBRECUR` alters frequently between taken and not-taken, a pattern which can be detrimental for the saturation counter predictors (e.g. T-NT-T-NT-... yields zero hits).

Our branch predictor also performs badly on indirect jumps such as JPR or JRL, as their branch target is highly dependent on the register content. The $2 register, which is the only register used for JPR in this workload, contains the function return address which changes very frequently because

of the tree recursion used in the Fibonacci function. As such, the branch predictor misses on most of the `JPR` instructions, which degrades the performance.

3.3.1 Structure of the branch predictor

The BTB contains 256 entries and is indexed by 8 LSB bits taken from the PC. As a small test, the prediction accuracy for different BTB index bit widths are measured (hysteresis counter).

BTB index width	Hits	Misses	Total	Accuracy
9 bits	160	111	271	0.590
8 bits	160	111	271	0.590
7 bits	25	246	271	0.092
6 bits	25	246	271	0.092
1 bit	7	264	271	0.026

The abrupt accuracy drop starting from 7 bits tells that at most 8 bits of the PC varies in this workload, and setting the index bit any more than that is not going to be beneficial for this workload.

3.4 Module parameters

The implementation can be configured to have data forwarding enabled/disabled or branch predictor changed by setting module parameters declared at the top of `cpu.v`.

4 Discussion

The result has shown in reality (in simulation) that the performance of the pipelined CPU could be vastly improved by utilizing common techniques learned in the class, such as data forwarding and branch prediction. Overall, the last version (hysteresis counter) gained speed improvement of 1.155 over primitive stall-based control hazard resolution, 1.818 over primitive data and control hazard resolution and 3.007 over the multicycle implementation.

Implementing all of this by hand was very helpful in understanding each techniques more in-depth and how they fit with each other. However, the evaluation of the implemented CPU lacks generality as the results may vary significantly for workloads other than the given test program. Therefore, for better comparison between different hazard resolution methods, the CPU must be evaluated against several different workloads.

Also, there may be various correctness issues remaining in the implementation that should be fixed, such as data forwarding not working without self-forwarding RF. The target address of the JPR instruction not being forwarded and thus requires pipeline stall is another such example.

5 Conclusion

Better understood why pipelining improves the performance of a CPU, what the control/data hazards are and how to resolve them, and successfully designed and implemented a pipelined CPU.