Lab06. Pipelined CPU

전기·정보공학부 김선우 (2019-16380)

1. Introduction

본 실습에서는, Pipelining이 구현된 CPU를 설계하고 Verilog로 구현한다. TSC ISA의 총 23개 명령어를 실행하는 microarchitecture를 설계하되, "IF->ID->EX->MEM->WB"의 단계 사이에 latch를 추가해 전체 throughput을 증가시킨다. Pipeline stall, Branch prediction(always taken)을 지원하는 baseline CPU를 설계하고, Data forwarding과 Advanced branch predictor(2-bit counter based)를 사용하는 CPU를 구현해 baseline CPU와 성능을 비교한다.

구현해야 할 것은 다음과 같다. (우선 1번을 구현한다.)

1) **Baseline CPU**: Branch, J, JR을 포함한 TSC ISA 명령어 전체를 IF->ID->EX->MEM->WB의 5단계 Pipelining으로 실행하는 microarchitecture를 설계한다. 고려할 Hazard는 Data Hazard와 Control Hazard가 있다. Hazard detection unit, Branch target buffer를 설계해 구현하고, testbench의 실행 사이클 수를 측정한다. Stall, Flush를 구현하며, Branch resolution은 ID 단계에 일어나도록 한다.

2) **Branch Predictor**: Control Hazard로 인한 Flush로 발생하는 사이클 손실을 줄이기 위해, Baseline의 "always-taken" prediction을 개선한다. 2-bit Predictor를 설계해 구현하고, testbench의 실행 사이클 수를 측정해 Baseline CPU와 비교한다.

각 실행 단계 사이의 latch는 clock에 동기적으로, 단계를 진행할 때마다 값을 업데이트한다. 한 명령어가 실행될 때, 이후 단계에 사용될 data 값을 latch로 넘겨주는 Datapath를 추가로 설계해야 한다.

실습을 통해 Pipelined CPU의 동작을 구체적으로 이해하고, Data forwarding과 Advanced branch prediction이 성능을 개선하는 정도를 실험해본다. Pipelined CPU가 Multi-cycle CPU에 비해 개선된 성능을 보이는 이유를 이해하고, 발생하는 Data, Control hazard와 해결 방법을 이해할 수 있다.

2. Design

모든 23개 명령어는 Pipeline에 따라 IF->ID->EX->MEM->WB의 5개 단계를 순차적으로 실행한다. 각 단계에서 실행해야 하는 조작에 따라 data와 control signal을 pipeline을 따라 전달해야 하며, 명령어 실행을 각 단계별로 나누어 이해해야 한다.

우선, Register File, ALU, Output Port, Data Memory interface를 제어하기 위해 필요한 control signal은 다음 Table 1과 같다. 이것은 지난 실습의 Multi-cycle CPU 설계와 유사한 구성이다.

Table 1 Control signals & used Stages (for RF, ALU, Output Port, Data Memory interface)

Stage (used at)	Control signals
ID	RegDst
EX	OutWrite, AluSrcB, AluOp
MEM	MemRead, MemWrite
WB	RegSource, RegWrite

그러나, Pipeline 구현을 위해 본 실습의 CPU에는 각 단계 사이에 Latch가 추가되어야 한다. 또 pipeline 구현의 특성상, data dependency가 있는 명령어가 나중 단계를 실행하고 있다면 ID 단계의 명령어는 ID에 멈춰 dependent한 명령어의 실행이 완료되기를 기다리고, 다음 명령어가 fetch되는 것을 중지해야 한다. Jump나 Branch 명령어의 다음 명령어를 always-taken 방식으로 fetch했다면, 실제 다음 명령어를 구하고 틀렸을 때는 이미 fetch한 명령어를 flush하고, 옳은 주소에서 다시 fetch하도록 해야 한다. 그리고 CPU의 실행을 중단하는 명령어인 HLT를 실행할 수 있어야 하며, 디버깅을 위해 실행한 명령어 개수를 세는 num_inst 포트의 값을 계산하는 부분도 필요하다. 따라서 pipelined CPU의 동작을 위해선 latch, data/control hazard detection, branch resolution, machine halt, instruction count를 위한 hardware와 control signal, datapath가 추가로 필요하다. 이러한 기능을 지원하는 microarchitecture의 회로도를 다음에 나타냈다. (Figure 1)

회로도를 그려 각 기능이 수행되어야 할 단계와, 필요한 control signal 및 datapath를 판단한다. 회로도를 그리며 Hazard detection, Branch target buffer, Control module의 logic을 설계할 수 있다.

다음 회로도는 Baseline CPU 기준이며, 이후 Forwarding, Branch predictor를 추가할 때마다 추가되는 회로를 설명한다.

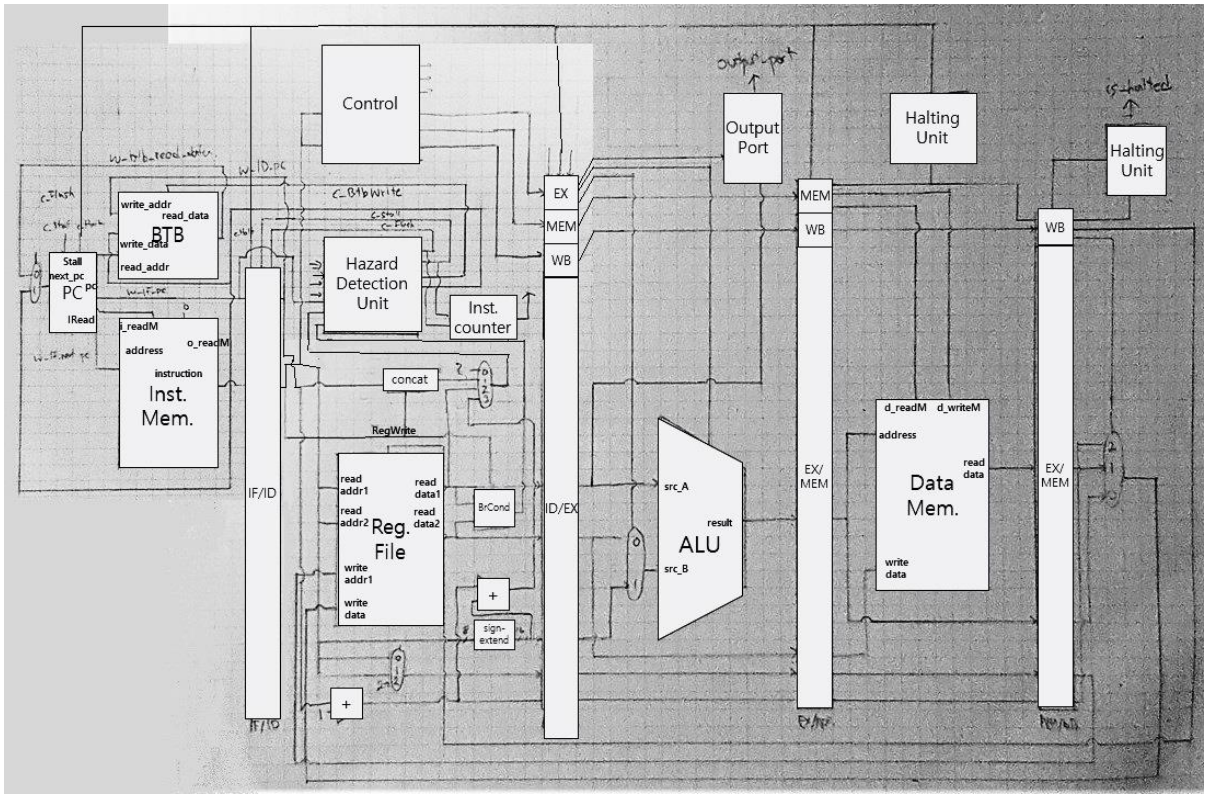

Figure 1 Circuit Diagram of Pipelined CPU module

1) Hazard detection and stall, flush

1. Data hazard

   Data dependency는, ID 단계에 있는 명령어가 $rs의 값을 사용하는데 EX, MEM, WB 단계에 있는 명령어들 중 $rs에 값을 쓰는 명령어가 있을 때 발생한다($rt에 대해서도 마찬가지). 따라서 detection을 위해 필요한 정보는 1) ID 단계의 명령어가 $rs, $rt를 사용하는지, 2) EX, MEM, WB 단계의 명령어가 Register File에 값을 쓰는지, 3) EX, MEM, WB 단계의 명령어가 값을 쓰는 주소가 $rs, 혹은 $rt인지이다. 1)의 정보는 명령어에 의존하므로 Control 모듈이 확인해서 값을 전달해 주도록 하고, 2), 3)은 각 단계의 RegWrite control, Reg write address data signal을 가져오면 된다.

   Detection의 결과 dependency가 하나라도 발견된다면, EX 단계로의 control signal은 0을 보내고, PC와 IF/ID latch가 값을 업데이트하는 것을 막는다. Stall signal을 출력하고, stall에 따르도록 PC, IF/ID latch의 동작을 설계한다.

2. Control hazard

   Control hazard는 다음 fetch한 명령어의 주소와, ID 단계에서 resolve된 옳은 다음 주소가 일치하지 않을 때 발생하며, 이때는 fetch한 명령어를 flush하고 IF 단계의 PC 값을 수정, BTB에 옳은 값을 써 주는 조작이 필요하다.

ID 단계에서 간단한 연산(add, concat 등 사용)을 이용해 Jump나 Branch의 target address를 구할 수 있고, RF 값을 기준으로 branch condition도 구할 수 있다. Control hazard detection을 위해 그렇게 구한 target address, branch taken, IF 단계의 명령어 주소 정보가 필요하다. 두 주소가 불일치한다면 IF 단계의 명령어를 flush하기 위해 IF/ID latch를 flush하고, BTB에 옳은 값을 쓴다. Branch 명령어의 경우, 최초 이후의 not-taken이라면 target address를 IF 단계에서 fetch할 때 miss가 발생함에 유의한다.

2) Branch target buffer

Reset시 BTB의 모든 entry를 "주소+1"로 초기화한다. Read, write를 지원하며, Control hazard 발생 시 Detection unit에서 write, write_data signal을 받아 값을 수정한다.

Branch predictor 를 구현할 때, BTB 를 확장하는 방식으로 구현한다. Per-address/Global history(shift), counter(saturation, hysteresis)와 같이, 선택할 수 있는 가짓수가 많다. 가능한 경우를 실험해보며 testbench 명령어 구성을 가장 잘 예측하는 모델을 정한다.

3) Control module and control signal pipelining

Control module은 ID 단계에서, fetch된 명령어에 대응하는 control signal을 출력해 적절한 실행 단계까지 전달하도록 한다. Table 1의 control signal들은 그대로 각 단계까지 전달되도록 pipeline을 구성한다. Table 1에 생략된 것으로는 HLT 명령어의 실행과 num_inst 포트 값의 계산이 있다.

HLT 명령어는, 앞 명령어가 모두 실행된 이후 CPU의 동작을 멈춰야 하므로 WB 단계의 시점에서 is_halted에 High를 전달한다. Pipelined CPU는 PC와 각 단계 사이의 latch들이 clock 신호에 동기적으로 값을 업데이트하면서 동작하므로, 각각의 업데이트를 멈춤으로 Halt를 구현할 수 있다. 이를 위해 PC와 모든 latch가 Halt 입력에 따라 업데이트를 멈추도록 설계하고, Halting unit을 만들어, HLT 명령어에 따른 Halt signal을 받을 시 모두에 Halt 신호를 전달하도록 한다. HLT 명령어가 WB 단계가 되는 시점에 CPU가 멈추게 하려면, MEM 단계에서 Halting unit이 동작해야 한다. MEM/WB latch만 WB 단계의 Halt signal을 인가하고, 나머지 PC 및 latch에는 MEM 단계의 Halt signal을 인가한다.

Num_inst를 계산하기 위해, Stall, Flush, Halt가 High인 경우가 아니면 매 사이클 ID 단계에 새로운 명령어가 들어옴을 이용한다. 디버깅을 위해, testbench에 따라 초기값을 조정하거나 num_inst가 일부 사이클에서만 유효한 값이 출력되도록 할 수 있다.

3. Implementation

Design 단계의 설계를 따라, 각 단계 사이의 latch에서 pipeline할 data, control은 다음 Table 2와 같다. 이러한 값들을 동기적으로 latching하며, stall/flush/halt에 적절히 반응하는 latch들을 각 단계 사이에 배치한다. ("Controlled by" 열에서 clock, reset_n은 제외하고 작성함)

Table 2 Pipeline latches and PC

Names (stages)	Control signals	Data	Controlled by
PC	-	[15:0] pc(inst. address)	Stall, Halt
IF/ID	-	[15:0] inst., [15:0] pc	Stall, Halt, Flush
ID/EX	Controls for EX, MEM, WB	[15:0] RF data A, B, [15:0] sign-extended immediate, [1:0] RF write addr, [15:0] pc+1	Halt
EX/MEM	Controls for MEM, WB	[15:0] alu result, [15:0] RF data B, [1:0] RF write addr, [15:0] pc+1	Halt
MEM/WB	Controls for WB, Including Halt	[15:0] memory read data, [15:0] alu result, [1:0] RF write addr, [15:0] pc+1	Halt

Latch, BTB, Hazard Detection unit, Instruction counter, Halting unit 및 ID 단계에서의 Branch resolution 회로를 추가하고, signal을 운반하는 모든 wire에 이름을 붙인 회로도는 다음과 같다.

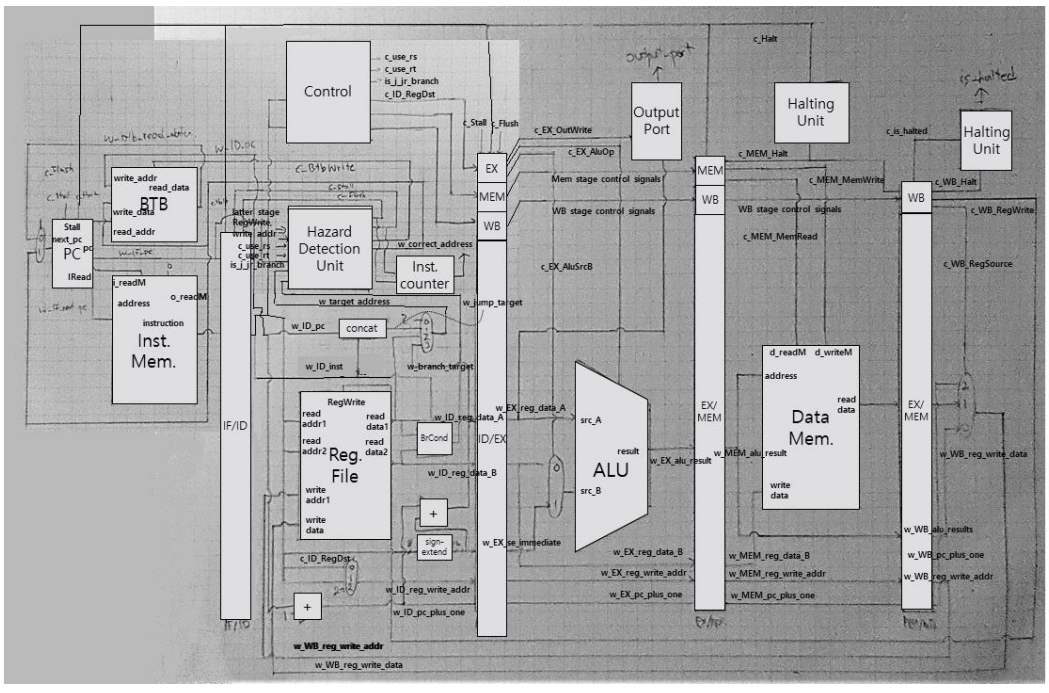

Figure 2 Circuit Diagram of Pipelined CPU module, all signals named

다음은 Control, Hazard Detection unit, BTB의 구현에 대한 설명이다.

1) Control (and other Pipelined signals)

각 실행 단계에 필요한 Control signal을 일시에 ID 단계에서 OPCODE, FUNCTION CODE에 따라 생성한 뒤, Stall/Halt를 따르는 각 pipeline register(latch)를 통해 사이클에 맞게 각 실행 단계로 전달한다. 앞 Design 단계에서 언급했듯이 Halt는 MEM 단계에서 MEM/WB latch를 제외한 모든 latch로 인가되지만, WB 단계로도 전달되어야 output port is_halted가 옳게 할당됨에 유의한다.

Control signal 외에도, 각 단계에서의 연산을 위해 필요한 data signal 또한 마찬가지로 pipeline한다. 이는 위 Table 2에 나타낸 것과 같다. 각 signal wire는 control인 경우 prefix "c_", data인 경우 "w_"를 붙였으며, 단계의 구분이 필요한 경우 "w_IF_inst", "w_ID_inst"처럼 단계의 이름을 붙여 구분했다. signal의 선언과 할당은, 단계별로 cpu.v 모듈에 작성했다.

2) Hazard Detection Unit

Data, Control Hazard를 detect하기 위해 필요한 모든 signal을 입력 받고, Stall, Flush, BTBWrite, correct_address, ID_branch_taken를 출력한다. Data Hazard은 EX, MEM, WB 단계의 dest reg 주소와 RegWrite signal 값을 기준으로, ID 단계의 source reg 주소와 비교 한다. 이때 ID 단계의 명령어가 어떤 source register를 사용하는지 등 명령어-specific한 정보는, Control module이 확인해 전달하도록 한다. Hazard시 Stall=High를 출력한다.

Control Hazard의 경우 IF-ID 단계 사이에서 대응하는 값을 비교해 Flush, BTBWrite signal을 출력하고, correct address를 계산해 IF로 넘긴다. IF 단계에서는, Flush가 발생할 시 PC 값을 BTB가 아닌 correct_address에서 받아오도록 하고, 같은 사이클에 BTB write도 실시한다. Design 단계에서 상술했듯, J나 JR과 달리 Branch 명령어는, not-taken 경우도 있음을 유의하고 signal의 logic을 결정한다.

3) BTB

256 * 16-bit 크기의 internal reg을 선언하고, 각각 address + 1로 초기화했다. 2-bit Per-address Counter를 구현하기 위해선 16-bit 폭을 18-bit로 확대했다. BTB에 값을 쓰는 BTBWrite signal과 target address는 Hazard detection unit에서 생성한다.

Branch prediction을 구현하고 accuracy를 측정하기 위해, decode된 branch 명령어와 rm miss/hit을 count해야 한다. 이때 Flush, Stall 동작으로 decode 단계에 한 branch 명령어가 여러 사이클 머무를 수 있으므로, 단순히 ID 단계에 branch 명령어가 들어온 사건이 아닌, Flush와 Stall signal을 고려한 것으로 branch count를 실시한다. Branch miss도 마찬가지로 계산한다. ID_inst == Branch이고 Flush == High인 것이 일차적으로 branch miss이나, Stall/Flush에 따라 중복되는 값이 없도록 계산한다.

4. Discussion

Baseline CPU를 구현해 testbench를 "All Pass"했고, 이 구현을 기준으로 실험을 진행했다.

이때 총 branch 개수는 92개, miss가 47개로 always taken의 accuracy = 49%였다.

최초에는 forwarding 없이, 구현한 BTB의 각 row에 2-bit을 추가해 Per-address counter-based, history-based를 실험해보고, 2-bit global history에 대해 branch prediction을 구현했다. 그 결과는 다음 표와 같은데, 유의미한 성능 개선이 관찰되지 않았다.

Method	**Baseline**	**2-bit local Hysteresis**	**2-bit local Saturation**	**2-bit local history, 11/10**	**2-bit local history, 10/01**	**2-bit global history, 11/10**	**2-bit global history, 10/01**
#Cycles	1993	1997	1997	1997	1993	1997	1993
Accuracy	49%	38%	38%	38%	49%	38%	49%

이는 testbench의 branch 명령어 간에 locality나 global correlation이 크지 않기 때문인 것으로 보인다. 특히 history-based prediction에서, per-address, global 모두 10/01처럼 값이 변할 때 taken으로 예측한 것이 11/10보다 accuracy가 큰 것도 이를 뒷받침할 수 있다.

Forwarding unit을 구현해 Stall 사이클 손실을 줄일 수 있다. 구현한 microarchitecture에서 RF 값이 사용되는 곳은 branch condition에 따른 taken 여부와 JR 명령어의 target address를 구하는 ID 단계, ALU 연산을 하는 EX 단계, 메모리 접근 주소와 store 값을 구하는 MEM 단계가 있다. 또 RF 값이 생성되는 곳은 ALU 연산 결과가 발생하는 EX 단계, 메모리의 값을 읽어오는 MEM 단계, Write-back이 실행되는 WB 단계가 있다. 각각 RegWrite signal과 dest/source register 주소를 비교해, Hazard detection 중 Stall signal을 구한 방식과 동일하게 값을 bypass, forwarding하는 logic을 구성할 수 있다. 이때 Hazard detection unit의 Stall signal 조건이 변화하며, 이는 모든 forwarding을 구현했을 때(WB internal forwarding, clock의 앞 절반에 write하고 뒤 절반에 read하는 scheme으로 Register File 구현), ID<->LW 단계 사이의 source/dest register에 dependency가 있는 경우로 유일하겠다. 이렇게 구할 경우, 대부분의 Stall loss가 제거되어 성능이 유의미하게 개선될 것이다.

5. Conclusion

Baseline CPU를 구현해, 제시된 testbench에 대해 All Pass의 결과를 얻었다.

최종 983개의 명령어를 1993 사이클만에 실행해, 평균 CPI = 2.03을 달성했다.

이는 Multi-cycle CPU의 CPI가 0.31이었던 것에 비해, 같은 clock 속도에서 speedup=550% 개선된 것이다.

Branch prediction을 통한 성능 개선은 유의미한 결과를 얻지 못했으나, local/global, counter/history-based 등 다양한 design decision을 구현하고 실험해 보았다.