**컴퓨터구조 2차 과제: 5 Stage Pipelined ARM-reduced Processor 분석**

2016130927 박준영, 2019320075 황도준

**1. Pipeline armreduced module**

Single에서와 달리 5 stage pipeline에서는 각 단계에 해당하는 레지스터 파이프를 통해 각 클럭을 통해 들어오는 instruction을 input을 받아들이고 output을 다음 파이프로 내보낸다. 이때 프로세서 하나에 여러 개의 instruction이 작동되고 있기 때문에 instruction이 레지스터에 write/read해야 하는 시점에 hazard가 일어날 수 있다. 뿐만 아니라 branch 등으로 인해 프로그램의 컨트롤이 바뀌면 파이프라인에 남아 있는 실행하지 않아야 하는 instruction이 실행되는 경우가 있을 수 있다. 이를 위해 data/control hazard에 알맞은 stall/forward를 넣어 방지했다.

또한 이 Timer의 동작 구현 상 instruction 21-24 루프를 통해 timecounter.v 내 output인 StatusR의 값이 변화하여 24번 isntruction의 branch로 분기할 수 있는 필요조건이 갖추어진다. Simulation tool 설정상 시간이 오래 걸리기 때문에 임의로 inst.mif의 22번 CMP insturction의 imm value를 1이 아닌 0으로 변경함으로써 간단히 single에서의 변화 형태와 비교한 timing simulation 정보를 통해 HEX/Register 값 등을 참조하여 코딩하였다.

하지만 Inst.mif을 원상태로 복구한 뒤 보드에 run한 결과 A-:00:00:00에 멈춰 있는 채로 동작하지 않았다. 따라서 armreduced 내 자체에서가 아니라 주변 TimerCounter.v 모듈에서 들어오는 StatusR 조건에 따라 동작 수행 여부가 결정된다고 추론하였고, TimeCounter.v의 CounterR 값을 ~reset/StatusR[0]값이 1이 될 때 원래 주던 0에서 32'h0x5F값을 주었다. 이때 정상적으로 보드에 run하지 않았던 경우와 달리 타이머가 작동하는 것을 확인하였는데, clk 타이밍에 따른 StautsR값 변화 때문으로 추측된다. 자세한 시뮬레이션 결과 및 보드 동작은 이후 timing simulation screen shot과 참조한 동영상을 통해 확인하기로 하겠다.

**2. Main-module (armreduced module)**

본 프로젝트를 구성하는 핵심은 ARMCPU.v 내의 armreduced 모듈로서 5 stage pipeline이라는 본 프로젝트 목적에 따라 IF, ID, EXE, MEM, WB 5개의 stage로 나누어져 진행되고 있다. 다음은 그 단계에 따라 어떤 동작이 이루어지고 있는 main module의 관점에서 분석한 결과이다.

1) IF stage

IF stage의 명령어는 파이프 레지스터 RegisterWBIF에서 나온다. 여기서 나온 명령어는 armreduced 모듈의 아웃풋이 되어 메모리의 주소로 동작한다. 메모리에서 주소에 맞는 명령어를 가져오면 그것이 RegisterIFID로 들어간다.

2) ID stage

ID stage에서는 크게 control unit을 통한 instruction decoding과 register file 참조가 일어난다. 각각의 모듈에 대한 구체적인 설명은 submodule 파트에서 하기로 한다.

기존 single cycle 구현과 차이가 있는 점은, ReadAddr을 결정하는 MUX가 레지스터 파일 안에서 레지스터 파일 바깥(armreduced 모듈 내)로 나왔다는 점이다. 이것은 이후 hazard를 처리할 때 읽는 레지스터 주소를 armreduced 모듈에서 넘겨주기 위함이다.

3) EXE stage

EXE stage에는 크게 condition unit과 ALU가 존재하는 것으로 볼 수 있다. ALU에 들어가는 값은 포워딩을 위한 MUX와 레지스터 파일에서 나온 값 혹은 immediate value를 선택하는 MUX를 통해 결정된다.

명령어의 Condition bit에 따른 명령어의 수행 여부가 여기서 결정된다. 구체적인 결정 방법은 Condition unit을 설명할 때 같이 설명한다.

4) MEM stage

MEM stage는 데이터 메모리를 참조하는 stage이다. 참조하고자 하는 데이터 주소가 armreduced 모듈의 아웃풋으로 나가며, 이에 상응하는 데이터가 데이터 메모리로부터 모듈의 인풋으로 들어온다.

5) WB stage

WB stage는 레지스터 파일에 연산된 결과를 쓰는 단계이다. 쓰고자 하는 레지스터 파일의 주소는 명령어를 통해 결정되며, ID stage부터 WA3D, WA3E, WA3M을 거쳐 WA3W로 흘러왔다. 쓰고자 하는 데이터 ResultW는, 1. 메모리에서 나온 데이터(ReadDataW), 2. ALU의 연산 결과(ALUOutW), 3. branch link에서 link되는 주소(pcoutW) 셋 중 하나가 선택된다. Branch and link의 구체적인 구현은 잠시 후에 설명한다.

6) Branch and Branch link

Branch instruction 중 funct[4]가 1인 경우에는 레지스터 파일의 $14에 링크로 현재 실행하고 있는 instruction 주소+4를 걸어주어야 한다. 본 구현에서는, 레지스터 파일 $15에 들어있는 현재 주소 값을 pcout으로 내보내 이것을 WB단계까지 흘려보내고, branch인 경우에는 MUX를 통해 원래 있던 address값이 아닌 14로 Writeaddress를 설정하는 방식을 택했다. branch 및 branch link 명령어는 다른 명령어에서 사용되는 WriteAddress 위치의 bit를 write를 위해 사용하지 않기 때문에 이 값을 임의로 14와 pc값으로 지정할 수 있다. 만일 Link라면 WB 단계 이후 Register에 값을 기록하지만, 일반적인 branch라면 값을 기록하지 않고 pc값만 변경해준다. 구체적인 로직은 다음과 같다.

1. ID stage에서, 레지스터 파일의 15번 레지스터에 들어있는 PC 값+4를 받아(pcoutD) EXE, MEM, WB stage로 파이프라인 레지스터를 거쳐 흘려보내준다.

2. EXE stage에서, 명령어가 branch이고 taken된 경우(i.e., BranchTakenE == 1), write back하는 레지스터 파일 주소를 14로 한다.

3. WB stage에서, 명령어가 branch이고 taken된 경우(i.e., BranchTakenW == 1), write back하는 데이터를 pcoutW으로 한다.

4. 만약 branch and link인 경우, 레지스터 파일의 $14에 pcoutW가 쓰여진다. 만약 branch without link인 경우, 레지스터에 쓰기가 enable되지 않기 때문에 아무것도 쓰여지지 않는다.

7) NextPC 값 결정

본 구현에서, 현재 IF stage에 있는 명령어의 주소는 레지스터를 통해 구현한 RegisterWBIF module에서 출력된다. RegisterIFID는 매 negative edge에 NextPC라는 input을 받아들여 reset/StallF 값 여부에 따라 PCF를 출력한다. 즉, NextPC는 MUX를 통해 세 가지 중 하나를 택하는데, 이는 reset이나 stall되지 않았을 때의 값이다. 이를 input으로 RegisterWBIF가 받아들여 output으로 출력하는 PCF가 RegisterFile의 input으로 입력되는 값이다. 자세한 과정은 다음과 같다.

RegisterIFID에 들어오는 input NextPC 값은 다음 세 가지 경우 중 하나가 MUX로 결정된다.

1. Branch가 taken된 경우: ALU에서 연산한 명령어 주소값으로 이동(ALUResultE)

- EXE stage의 명령어가 branch인 경우에 BranchE의 값이 1이 되고, condEx 값이 각 NZCV signal 값에 따라 실행 여부를 결정한다. 즉 branch이면서 동시에 condEx == 1’b1이면 branch를 실행해야 하는 경우임을 알 수 있고, 이를 BranchTakenE라는 wire을 통해 알 수 있다. 브런치 target 값은 EXE 단계의 combinational logic인 ALU operation을 통해 결정되므로, 다음 NextPC값은 바로 ALUResultE 값이 된다.

2. MOV $15인 경우: WB 단계에서 레지스터 파일에 쓰는 값으로 이동(ResultW)

- 이 경우는 사실상 MIPS의 jump instruction과 같은 것으로 볼 수 있는데, R15를 대상으로 하는 MOV 명령어만을 CtrlSig 값 할당 이후 PCSrc값을 1로 변경해주어 타 instruction 경우와 차별화해주었다. 즉 ResultW가 NextPC로 선택되는 경우는 이 단 한 가지의 경우이다.

3. 일반적인 경우 – 순차 이동: PC+4로 이동(PCPlus4F):

- PC 컨트롤을 이동하지 않을 경우 PC값은 자연스럽게 원래 PC값에 32’d4를 해준 값이 할당된다.

이 NextPC와 현재 RegisterWBIF가 가지고 있는 기존의 PC값, 즉 PCF 중 어떤 값을 선택할지는 그 단계에 있는 Stall의 여부와 관련이 있다. 물론 여기에서도 reset일 때 모든 데이터 값을 0으로 만드는 과정 역시 포함되어 있다.

MUX에서는 상기한 세 가지 경우 중 하나를 선택하지만, stall을 고려할 때 다음 사이클에도 기존 PCF를 유지하는 경우가 있다. Single이 아닌 pipeline 구조이기 때문에 필연적으로 data/control hazard가 일어날 수밖에 없는데, 특히 IF단계에 적용된 Stall은 다음 값으로 할당받은 NextPC가 아니라, 지금 가지고 있는 PC를 다시 한 번 반복한다. Output을 출력하는 본 module이 Register 구조이므로 PCF는 한 클럭 전의 값을 보존할 수 있고, StallF, 이 모듈에서는 en이라고 명명된 신호가 0이 되면 NetxPC가 아닌 PCF를 출력한다.

이러한 과정을 통해 총 4가지 경우 중 다음에 들어올 PC 값을 결정할 수 있다.

**3. Sub-modules**

1) Hazard unit

Pipelined processor는 구조의 특성상 hazard를 처리해야 한다. 본 프로젝트에서는 hazard 중 data hazard를 stall과 forward를 이용하여, 또 control hazard를 stall을 이용하여 처리하는 것을 목표로 한다. 이에 따라 본 프로젝트에서는 3개의 모듈 StallforControlHazard, StallforDataHazard, ForwardforDataHazard이 hazardunits.v 안에 구현되어 있다. HazardDetection과 종류에 알맞은 대책을 각 케이스에 따라 구현하였다.

(1) StallforControlHazard

Control hazard는 PC의 control에 대한 hazard이다. 본 프로젝트에서는 branch not-taken 방법을 이용하여, Branch instruction이 taken되기 전까지는 pipeline을 통해서 새로운 instruction이 계속해서 들어올 수 있도록 설계하였다. 하지만 EXE단에서 Branch가 taken 여부가 결정될 때, 즉 control signal의 BranchTakenE 값이 0이 아닌 1이 될 때, FlushD와 FlushE 신호를 1로 하여 IF/ID와 ID/EXE 파이프 레지스터를 모두 초기화한다. 파이프 레지스터를 초기화함으로써 분기가 taken되지 않을 것으로 예상하고 수행하던 instruction을 안전하게 모두 0으로 flush하고, branch한 instruction의 동작을 수행할 수 있다.

(2) StallforDataHazard

Data hazard를 처리하기 위해서 본 프로젝트에서는 stall과 forward를 사용하였다. 본 모듈은 그 중 stall을 주기 위한 모듈이다.

stall을 판단하는 과정은 다음과 같다.

1. 만약 EXE단에 있는 명령어가 load이고,

2. EXE단의 명령어가 쓰고자 하는 레지스터(WriteRegisterE)와, ID 단의 명령어가 읽고자 하는 레지스터 두 개(ReadRegister1과 ReadRegister2) 중 하나라도 같다면

3. Stall이 들어가고(StallF = 1, StallD = 1), ID/EXE 파이프 레지스터의 컨트롤 시그널을 모두 0으로 초기화시킨다.

ID/EXE 파이프 레지스터의 컨트롤 시그널을 초기화시키는 이유는, stall을 주게 되면 현재 ID 단에 있는 명령어가 다음 사이클에 EXE로 넘어가지 않아야 하는데, ID/EXE 파이프 레지스터에는 stall을 위한 시그널이 따로 없으므로 ID stage에서 fetch된 레지스터 값들이 EXE단으로 다음 사이클에 넘어가게 되고, 이 때 컨트롤 시그널을 초기화시키지 않으면 MemWrite나 RegWrite등이 1로 되어 다음 stage로 넘어가 메모리와 레지스터 파일에 garbage 값이 쓰일 수 있기 때문이다.

StallF와 StallD는 1이 되면 각각 WB/IF와 IF/ID 파이프 레지스터의 write를 disable시키므로, 새로운 명령어가 들어오지 않고 현재 명령어가 그대로 머물 수 있다. 특히 이 경우 RegisterFile에서 input으로 받아들일 PC값을 출력하는 RegisterWBIF의 PCF output이 NextPC가 아니라 이전 PCF과 동일한 값을 출력함을 주목하라.

추가로 본 구현에서는 상기한 경우 이외에 stall이 들어가는 경우가 한 가지 더 존재한다. ID 단계에 있는 명령어의 읽는 레지스터 파일 주소와 WB 단계에 있는 명령어의 쓰는 레지스터 파일 주소가 같은 경우에 stall이 들어가도록 되어 있다. 이는 이론과 배치되는 부분이나, 프로젝트를 진행하면서 Forwarding이 일어나지 않는 정상적인 경우에 일어난 버그를 해결하고자 애드혹의 목적으로 들어간 것이다. 이 부분이 없는 경우 모종의 이유로 레지스터 파일에 값이 제대로 된 사이클에 쓰여지지 않는 것이 확인되었고, 그에 따라 Stall의 경우로 추가하였다.

(3) ForwardforDataHazard

Data Hazard를 해결할 때 Stall은 분명 유효한 대안이 될 수 있지만, 계속해서 hazard가 발생하는 가운데 stall은 파이프라인의 가장 큰 목적인 퍼포먼스 향상을 저해한다. 이에 따라 Stall이 필수적인 경우 (즉 EXE단계 명령어 LDR가 이전 ID단계 명령어가 Read하는 레지스터값을 write하는 경우)가 아니라면 EXE, MEM, WB 단계의 명령어 간의 Write/Read/RegWrite 값의 일치 여부를 확인함으로써 ‘레지스터 값이 write되기 전’ 시점에 유효한 값을 미리 전달해줄 수 있다. 이를 forwarding이라고 하며, 본 ForwardforDataHazard에서는 ‘어떠한 값을 MXU를 통해 넘길 것인가’, 즉 SrcA와 SrcB에 해당하는 값을 선택하기 위한 ForwardAE/BE을 출력하는 combinational logic이 구현되어 있다. 실제적인 값이 넘겨지는 과정은 main module에서 MUX를 통해 이루어진다. 구체적인 로직은 다음과 같다.

1. “match” wire에, EXE stage에 있는 명령어가 읽어낸 레지스터 주소 중 MEM stage나 WB stage에 있는 명령어가 write back하고자 하는 주소가 있는지 여부가 출력된다.

2. 만약에 MEM stage와 EXE stage가 겹친다면, MUX에 컨트롤 시그널 2’b10을 준다.

3. 만약에 WB stage와 EXE stage가 겹친다면, MUX에 컨트롤 시그널 2’b01을 준다.

4. 위의 경우에 해당되지 않는다면 MUX에 2’b00을 준다.

만일 EXE와 MEM이 겹치고, EXE와 WB 단계 또한 이중적으로 겹칠 때에는 어떻게 해야 하는가? 이 경우 일어날 수 있는 오류를 방지하기 위해 MEM 단계를 WB 단계보다 앞서 확인하고 있다. 이 경우 MEM 값이 더 최신 값이기 때문에, 결과적으로 EXE가 필요로 요구하는 값은 WB에서 쓰는 값이 아닌 MEM에서 쓰는 값이기 때문이다. 이러한 구조를 통해 ForwardAE/BE의 값이 결정되어, main module에서 사용할 Src 값이 결정된다.

2) Pipeline register : WBIF / IFID / IDEXE / EXEMEM / MEMWB

본 프로젝트에서는 piperegisters.v 파일 내에 5개의 파이프라인 레지스터가 구현되었다. 이름은 각각 RegisterIFID, RegisterIDEXE, RegisterEXEMEM, RegisterMEMWB, RegisterWBIF이다. 모든 파이프 레지스터들은 기본적으로 negative edge에 출력 값을 갱신한다는 점은 같다. 또 FPGA 보드 상에서 reset하는 경우에 대응하기 위해 레지스터 파일 뿐만 아니라 파이프라인 레지스터 또한 reset 시그널이 들어온 경우 모든 출력값을 0으로 바꾼다는 점은 같다. 하지만 각 파이프 레지스터마다 write를 enable/disable하는 신호나 flush하는 신호 등 필요에 따라 다른 signal을 받는 경우가 있으므로, 이런 경우가 있는 파이프 레지스터에 대해 추가적으로 설명한다.

(1) RegisterWBIF

본 레지스터는 input으로 en을 추가적으로 받으며 RegisterFile로 들어갈 PCF를 출력한다. 만일 data hazard가 존재한다면 en값을 통해 stall 여부를 확인할 수 있으며, 값이 0인 경우 negative edge에서도 새로운 값으로 PCF를 업데이트하지 않고 기존의 값을 유지한다. 이 과정은 앞서 main module의 다음 PC값을 어떻게 결정하는지를 서술하며 보다 자세히 다루었다.

(2) RegisterIFID

본 레지스터는 input으로 clr와 en을 추가적으로 받는다. clr은 control hazard가 있는 경우 stall하기 위해 사용되며, clr이 1인 경우 모든 출력값이 0이 된다. en은 data hazard가 있는 경우 stall하기 위해 사용되며, en이 0인 경우 레지스터는 입력으로 들어오는 값을 negative edge에도 받아들이지 않는다.

IFID 레지스터가 output으로 출력한 데이터는 곧 CtrlSig로 넘겨져 Decoding되고 RegisterFile로 넘겨지는데, 특히 기존에 RegisterFile 내에서 ReadAddr2를 읽었던 Single cycle implementation과는 달리 ReadAddr2를 RegSrc 값에 따라 미리 operand2 또는 operand3를 읽을 것인지 MUX를 통해 결정하고 있다. 이는 single에서 한 사이클 안에서 waddr와 addr2 중에 무엇을 읽을 것인지 정해야 했던 반면, 파이프라인 구조에서는 읽는 시점이 다르기 때문이다. 특히 waddr는 각 레지스터 파이프를 통해 WB까지 전달해주고 있다. 이밖에도 imm 연산을 위해 ExtendMux module에 input을 넣어주는데, 이는 single과 정확히 일치하고 있다.

(3) RegisterIDEXE

본 레지스터는 ID단계에서 해독한 컨트롤 시그널 정보와 RegisterFile에서 read한 데이터를 모두 받아들이는데, clr을 input으로 추가로 받는 게 특징적이다. Clr을 통한 동작은 RegisterIFID에서 서술한 바와 일치하는데, flushCtrilSig라는 input이 추가된다. 이 signal은 data hazard가 있는 경우 stall을 주기 위한 signal로 이 값이 1이 되면 본 레지스터에 들어온 값들 중 control signal들만 모두 0이 된다.

(4) RegisterEXEMEM

본 레지스터는 EXE단계에서 만들어진 컨트롤 시그널을 추가로 input으로 받으며 reset이 아닐 경우 동일한 값을 다음 MEMWB로 출력해준다. 본 설계에서 MEM 단계에 이루어지는 메모리 참조 과정은 기존 single cycle에서와 동일한 양상을 보인다. armreduced module 자체가 output으로 출력하고 있는 memaddr, writedata에 ALU연산을 통해 구한 메모리 주솟값과 그 값에 쓸 데이터를 assign함으로써 작동되는데, 그 여부는 memwrite에 MemWriteM을 할당함으로써 확인할 수 있다.

(5) RegisterMEMWB

본 레지스터는 EXEMEM 레지스터가 전달한 데이터를 input으로 받는데, armreduced module에서 데이터 메모리를 읽은 readdata가 추가된다. 만일 branch link로 인해 pc값을 기록해야 한다면 pcoutW가, 참조한 메모리에서 읽은 값이 선택되어야 한다면 ReadDataW가, 그리고 EXE단계 이후 연산한 ALU 값을 기록해야 한다면 ALUResultW가 각 컨트롤 시그널 값에 따라 MUX로 결정되는데, 이는 파이프 레지스터 바깥 전체 모듈 상에서 구현하였다.

3) Register File

RegisterFile은 프로그램 실행에 필요한 데이터들이 연산될 수 있도록 저장된 곳으로, ARMCPU.v 파일 내에 RegisterFile이라는 이름으로 존재한다. 본 구현에서는 구현된 single cycle과 같이 clock의 negative edge에 write를, positive edge에 read를 수행하도록 하였다.

MIPS와 비교하여 ARM의 두드러지는 차이점 중 하나는 ARM에서 $15가 PC로 사용된다는 점이다. RegisterFile은 PC값을 R15에 저장하여 출력, 만일 변경이 필요할 시 이를 write한다. 기존 Single cycle model에서는 다음 PC값을 받아 저장하지 않으면 이후 유의미한 PC 값으로 넘어갈 수 없는 반면, 현재 파이프라인 구조에서는 RegisterWBIF를 통해 이 역할을 대신 수행하고 있다. 따라서 기존 single의 pcout이 원래 지닌 의미는 전술한 레지스터 내에서 구현되고 있다. 이에 따라 본 설계에서 레지스터 파일 15번을 아웃풋으로 내보내는 pcout은 Branch link가 수행되어 R14에 Branch가 실행된 그 시점의 PC를 기록하기 위한 변수로 사용되고 있다.

Pipelining으로 stage가 나뉜 것에 대응하기 위해, register file이 받는 RegSrc 신호도 바꾸었다. ID 단계가 끝난 뒤 들어온 명령어의 RegSrc에 따라 값이 바로 쓰일 수 없기 때문에 레지스터 파이프에서 Single 구현과는 달리 RegSrc[0] 값을 output으로 계속 다음 레지스터로 보내고 있으며, 이 값을 WB단계에서 확인할 수 있을 때 write된다. 즉 RegSrcW를 통해 write되는 시점을 구별할 수 있는데, 이는 read에서는 불필요하므로 single 구현과 일치한다.

4) Combinational logic

이 부분에서는 control signal을 만드는 control unit과 조건에 따른 분기를 결정하는 conditional unit에 대해 설명한다. control unit은 큰 모듈인 ctrlSig와, 그 아래의 하위 모듈 2개 Decoder, ConditionalLogic으로 이루어진다. conditional unit은 conditionUnit 모듈로 구성된다.

(1) Control unit

ctrlSig에서 나오는 control signal output은 ALUOp, ImmSrc, RegSrc, PCSrc, RegWrite, MemWrite, MemtoReg, ALUSrc, Svalue, Branch 총 10개이다. 기존에 있던 모듈에서 프로젝트 수행 중 추가한 output은 Branch이다. Branch는 control unit으로 들어온 instruction이 branch인 경우 1을 출력한다. 이 output이 생긴 이유는, single cycle implementation과는 달리 조건을 판단하는 부분과 control signal을 계산하는 부분이 각각 EXE stage와 ID stage로 나누어졌기 때문에 control unit 내에서 자체적으로 조건을 판단할 수 없었기 때문이다.

(2) Conditional Unit

conditionUnit 모듈은 input으로 FlagWriteE, condE, FlagsE, ALUFlags를 받아 output으로 NZCV와 condEx를 내놓는다.

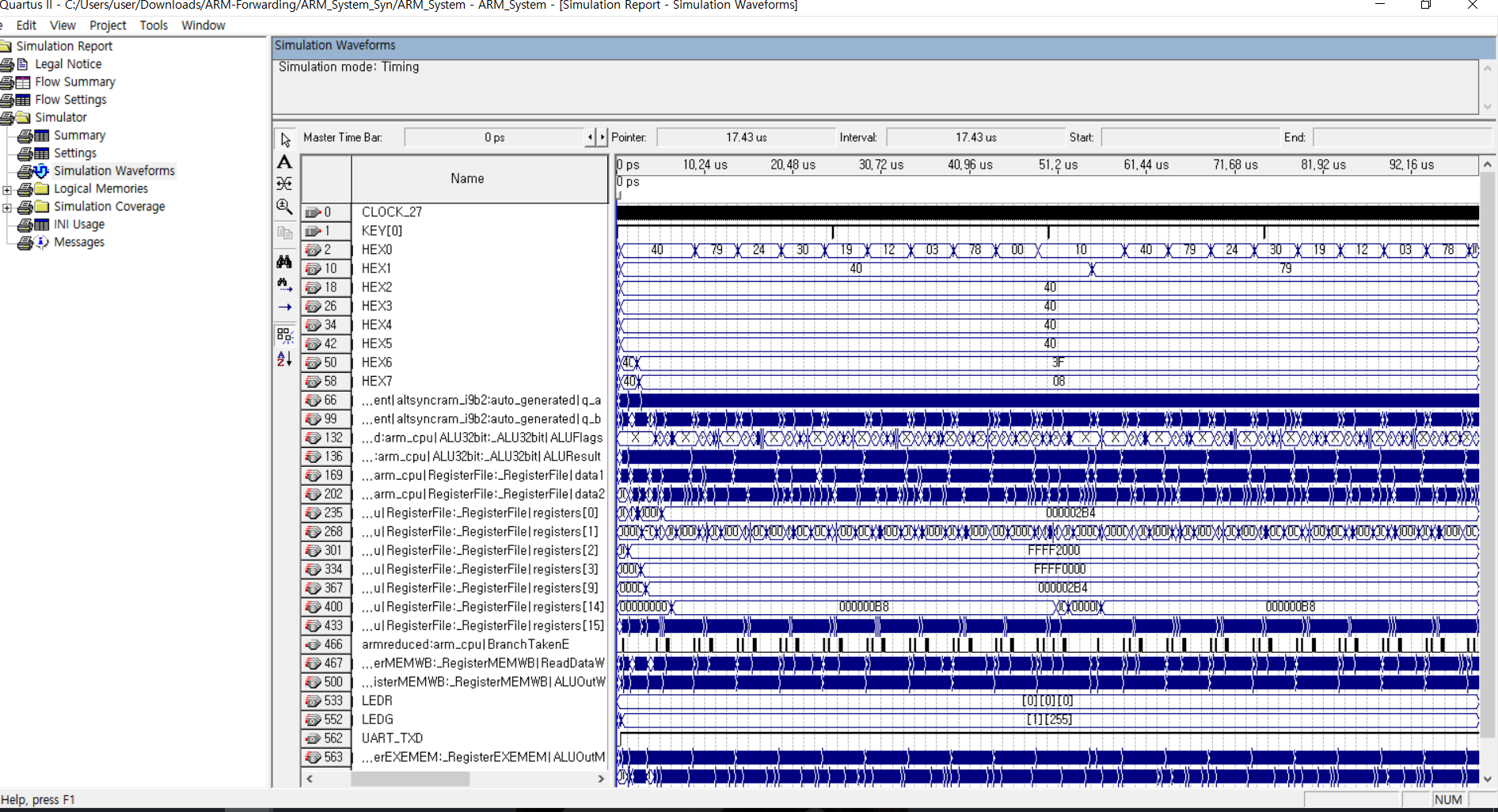
FlagWriteE는 control unit에서 나온 Svalue로 이것이 1이면 NZCV 값을 갱신한다. condE는 EXE stage에 들어와 있는 instruction의 condition bits이다. FlagsE는 이전 사이클의 NZCV 값으로, 본 구현에서 NZCV 값은 ID/EXE 파이프 레지스터에 저장되어 있기 때문에 그곳에서 나온 것이다. ALUFlags는 ALU에서 연산한 결과에 따라 결정되는 것으로, 본 프로젝트에서는 Z bit만 사용하기 때문에 그 부분만 구현되어 있다.

NZCV는 새로 결정된 NZCV bits 값으로, FlagWriteE가 1일 때에만 갱신된다. condEx는 NZCV bit 값이 들어온 condE 값과 일치하는지를 보고 일치하면 1을 출력한다. 이 값은 armreduced 모듈 내에서 PCSrcE, RegWriteE, MemWriteE, BranchE와 AND 연산을 하여 조건에 따라 명령어가 수행될 수 있도록 한다.

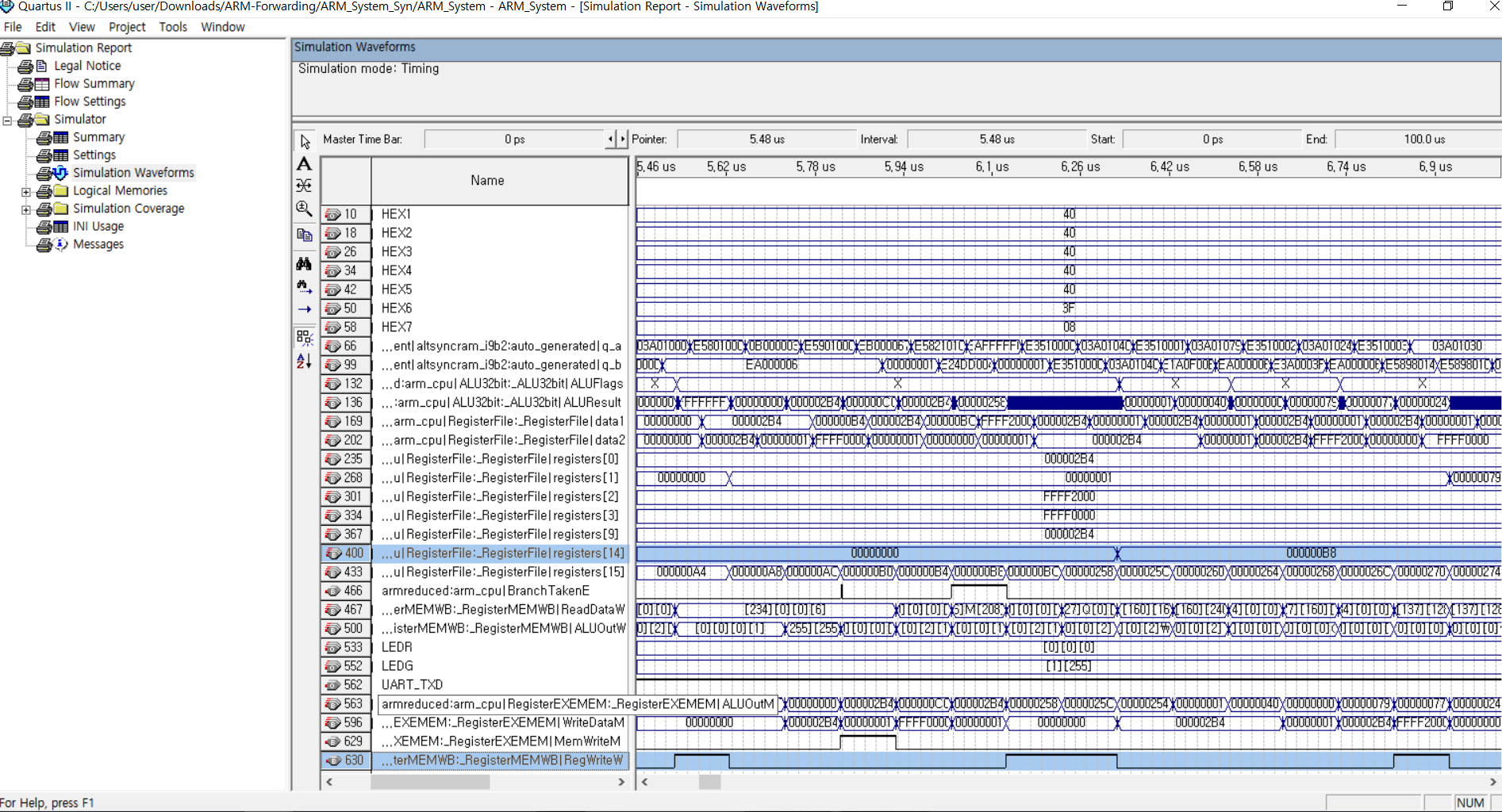
**4. Timing Simulation 및 설계 한계 추론**

이러한 코딩을 바탕으로 module을 설계하였고, loop에서 분기할 수 있는 요건을 마련한 후 얻어낸 시뮬레이션 결과값 역시 single cycle implementation에서의 HEX, Register value 등이 같아 보드 run 시에서도 동일한 동작을 보여줄 것으로 예상하였으나, (TimerCounter.v 값을 임의로 변경한 이후에도) 보드상에서 시간 부분에서 12에서 13…23으로 올라가지 않고 12에서 바로 00으로 초기화되는 오작동을 관찰할 수 있었다. 이외 분/초 부분에서는 정상적으로 작동하였음을 관찰하였다.

한편 다음은 loop를 inst.mif 명령어 상에서 탈출한 조건으로 재구성하였을 때 본 코드가 보이는 변화값이다. 그리고 simulation tool에서 end time 100us로 변경하여 기존의 25ns보다 더 긴 시간동안 시뮬레이션 형태를 기존 single과 비교하였다.



이 경우 HEX/Register 값 변화 측면에서 기존의 Single을 loop 탈출한 경우에서와 같이 시뮬레이션을 돌린 때와 동일한 양상을 보이고, 정상적으로 instruction 동작을 수행하고 있다고 판단했다. 다음은 특히 PC control이 바뀌면서 동시에 레지스터에 write를 해줘야 하는 Branch link할 시의 동작 과정이다.



Branch link 뿐만 아니라 LDR, STR 등 다른 instruction 또한 RegWrite 및 MemWrite를 통해 잘 동작하고 있는지 확인할 수 있었고, 특히 HEX 변화 측면에서 동일하기에 무리 없이 카운터 동작을 수행할 수 있으리라 판단했지만, CounterR 값을 임의로 변경하지 않고서는 보드 상에서 변화를 관찰할 수 없었다.