**Lab3 Report**

**20230642 이채영, 20230683 박한비**

1. Introduction

Lab3에서는 RISC-V instruction을 기반으로 하는 Multi Cycle CPU를 구현하였다. 하나의 cycle 동안 하나의 instruction을 실행한 Single Cycle CPU와 달리 Multi Cycle CPU는 각 instruction을 여러 단계로 나누고 단계별로 cycle을 할당하여 한 instruction을 실행하는 데에 여러 사이클을 거친다.

Multi Cycle CPU는 cycle을 더 작은 단위로 나누어 각 instruction을 실행하는 동안 해당 instruction이 필요한 만큼의 cycle, 즉 시간만 소비하도록 한다. 이러한 점 덕분에 가장 오래 걸리는 instruction의 실행 시간에 cycle을 맞추어 모든 instruction의 실행 시간이 가장 오래 걸리는 instruction과 같아지는 Single Cycle CPU보다 프로그램 실행 시간을 줄이고, 속도를 높일 수 있다.

1. Design

기본적으로 Lab3\_multi\_cycle\_cpu 교안의 8쪽에 제시된 회로도를 바탕으로 이전 Lab2에서 구현한 single cycle cpu 위에 추가적인 부분을 구현 및 수정하였다. PC, Memory, Control unit, Register file, ALU, Immediate Generator, ALU control unit와 2:1 Mux 4개, 4:1 Mux 1개 그리고 Instruction register, Memory data register, ALU out register, Read data 1/2 register 로 구성하였다.

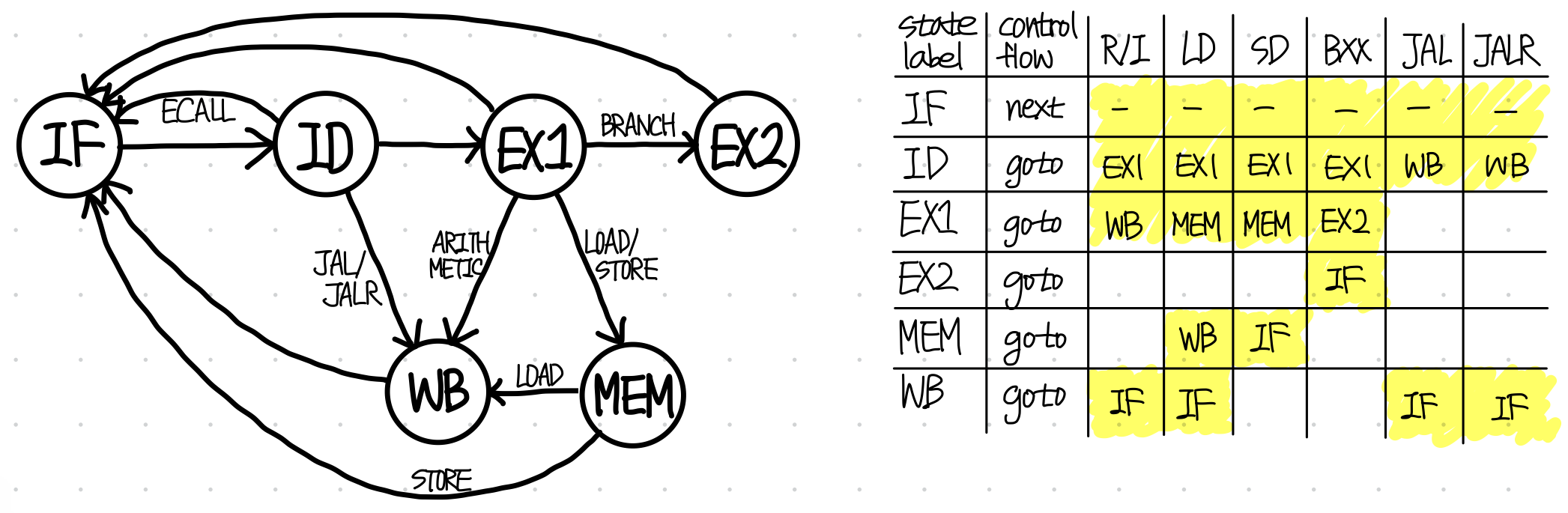
clk은 positive edge를 기준으로 동작하며, is\_halted 신호와 통합한 clk\_and\_ishalted를 사용하였다. clk을 받는 모듈(synchornous)은 PC, Memory, Control unit, Register file과 Instruction register, Memory data register, ALU out register, Read data 1/2 register이며 나머지 ALU, Immediate Generator, ALU control unit과 2:1 Mux 4개, 4:1 Mux 1개 는 asynchornous하게 작동한다. 아래 그림에 코드에서 사용한 모듈 이름을 빨간색 글씨로, wire의 이름(single cycle cpu와 달라진 부분 위주)은 파란색 글씨로 표기하였다.

아래 회로도를 보면 알 수 있듯이 Resource reuse를 위해서 Data memory와 Instruction memory를 Memory로, 각종 arithmetic 연산을 하는 resource들을 ALU로 병합하였다. Resource를 여러 용도로 재사용하기 위해 Memory와 ALU의 output을 저장해놓을 register들을 추가해주고(IR, MDR, ALUOut) 여러 용도에서의 input을 컨트롤하기 위해 Read data를 저장할 register들과 Memory 및 ALU의 input을 조정할 MUX들을 추가하였다.

텍스트, 도표, 평면도, 개략도이(가) 표시된 사진

자동 생성된 설명

Control unit을 위한 Microcode controller state design은 아래 그림과 같다.



왼쪽 그림은 FSM을 나타낸 것이다.

오른쪽 표의 노란색 표시는 각 instruction이 지나는 state를 나타낸 것으로, 각 스테이지는 하나의 state, 즉 한 사이클로 설계하였지만 Branch의 경우에만 EX 스테이지는 두 사이클로 매핑된다.

아래 그림은 resource reuse를 위해 한 스테이지 내에서 resource을 중복하여 쓰지 않도록 고려하며 각 instruction 마다 스테이지 별로 datapath를 배치한 표이다.

텍스트, 폰트, 라인, 스크린샷이(가) 표시된 사진

자동 생성된 설명

1. Implementation
2. cpu.v

Multi-cycle CPU를 구현한 메인 모듈로, 내부에 여러 모듈들을 include하고 wire로 적절히 연결하며 cpu의 회로를 구현하였다. 위의 design에서 제시한 설계도와 동일한 구조이다.

1. PC.v

텍스트, 스크린샷, 디스플레이, 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

pc(program counter)의 값을 업데이트하며 CPU가 현재(그리고 다음) 수행할 instruction line의 메모리를 가리키도록 하여 프로세스의 흐름을 통제한다.

* clock synchronous : clk이 들어올 때마다(positive edge) 다음 PC로 넘어갈 수 있는 상태인지, 즉 각 instruction의 마지막 state를 마친 상태인지를 pc\_write\_final을 통해 검사한다. 마지막 state를 마쳤으면 current\_pc를 next\_pc로 업데이트하여 CPU가 다음 instruction line을 수행할 수 있게 한다. reset 신호가 들어왔을 경우 pc를 0으로 초기화시켜 CPU가 다음 process를 실행할 수 있도록 해준다.

1. Memory.v

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

data를 저장한 magic memory를 지니고 있으며 input으로 들어오는 인덱스에 따라 따라 메모리에서 인덱스 주소에 해당하는 값을 출력하거나 입력한다. 메모리에서 인덱스 주소에 해당하는 값을 출력하여 레지스터에 저장하는 load instruction을 수행하거나 레지스터 값을 반대로 메모리에 입력, 저장하는 store instruction을 수행한다.

- clock asynchronous : clk이 들어오는 것과 관계없이 비동기식으로 input으로 받은 addr의 인덱스 위치에 해당 하는 메모리에서의 값을 output으로 출력한다. 단, 이는 mem\_read 활성화/mem\_write 비활성화된 경우에만 수행한다.

- clock synchronous : clk이 들어올 때마다(positive edge) input으로 addr의 인덱스 (기존의 skeleton code에 따라 addr 중 [15:2]만 참고) 위치에 해당하는 메모리에 서의 값을 input으로 받은 din 값으로 변경한다. 단, 이는 mem\_write 활성화/ mem\_read 비활성화된 경우에만 수행한다.

1. RegisterFile.v

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

32개의 register를 지니고 있으며 input으로 들어오는 인덱스에 따라 register의 값을 출력하거나 입력(이 경우 write 값도 필요)한다. 메모리에서 pc 주소에 해당하는 instruction을 출력하여 cpu가 해당 instruction을 해석하고 실행할 수 있게 한다. Single-cycle CPU에서 구현했던 register\_file과 동일하다.

* clock asynchronous : clk이 들어오는 것과 관계없이 비동기식으로 input으로 받은 rs1, rs2의 인덱스 위치에 해당하는 레지스터 값을 output으로 출력한다. 또한 is\_halted에 기본적으로 0을 할당하나 is\_ecall 신호가 들어왔을 경우 비동기식으로 1을 할당해준다.
* clock synchronous : clk이 들어올 때마다(positive edge) write\_enable이 1이라면 (레지스터에 WB을 할 때 활성화) input으로 받은 rd의 인덱스 위치에 해당하는 레지스터 값을 input으로 받은 rd\_din 값으로 변경한다.

1. ImmediateGenerator.v

텍스트, 스크린샷, 소프트웨어, 운영 체제이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

instruction에서 이용되는 immediate 값이 instruction format 상에선 나누어져 저장되어 있다. 이를 instruction 타입을 참고해 완전한 immediate 값으로 합쳐서 output 으로 출력해준다. Single-cycle CPU에서의 immediate\_generator와 동일한 모듈이다.

- clock asynchronous : clk이 들어오는 것과 관계없이 비동기식으로 instruction 값 을 input으로 받아 opcode를 추출하고 opcode에 따라 부분부분 저장된 immediate part를 적절하게 합쳐서 완전한 immediate를 만든다.

1. ALUControlUnit.v

Instruction의 타입에 따라 ALU.v에서 적절한 arithmetic 연산을 수행하도록 alu의 operation code를 설정해준다. 기존 Single-cycle CPU의 alu\_control\_unit에서 output을 alu\_select으로 바꾸고, input으로 2비트의 alu\_op를 추가하였다. alu\_op는 instruction 타입에 따라 ALU에서 덧셈, 뺄셈, ALU 또는 nop(연산 X)을 수행할지 결정한다.

* clock asynchronous : clk이 들어오는 것과 관계없이 비동기식으로 instruction 값의 일부인 opcode와 funct3, funct7\_5를 input으로 받아 그 값에 따라 적절하게 alu의 operation code(alu\_op)를 설정해준다. alu\_op의 인덱스상 [5:2] 위치는 arithmetic operation의 타입을 정해주고 [1:0] 위치는 instruction이 branch 타입일 경우 BEQ/BNE/BGE/BLT 중 어느 instruction인지 알려주는 역할(btype)이다.

1. ALU.v

CPU 상에서 arithmetic operation instruction과 그 외 instruction을 위한 arithmetic 연산을 수행한다. Single-cycle CPU에서 구현했던 ALU 모듈과 동일하다.

* clock asynchronous : clk이 들어오는 것과 관계없이 비동기식으로alu\_control\_unit.v에서 설정한 alu\_op를 input으로 받아 일부([5:2])를 function code로 설정, 일부([1:0])를 branch type으로 설정하고 function code에 따라 적절한 arithmetic 연산을 input으로 들어온 두 피연산자 A, B에 수행하여 나온 결과값을 output으로 출력한다. 또한 branch type에 따라 arithmetic 연산의 결과값에 다른 비교 연산자를 써서 조건문을 만들고 조건문을 통과할 경우 branch의 조건을 통과했다는 의미인 alu\_bcond 신호를 활성화해 output으로 출력한다.

1. ControlUnit.v

ControlUnit 모듈에서는 instruction에 따라 current state를 next state로 synchronous하게 업데이트하는 microcontroller와 state에 따라 control signal들을 활성화/비활성화하는 control unit의 기능을 수행한다. State는 IF, ID, EX1, EX2, MEM, WB 총 6개를 정의했다. Instruction에 따라 state를 변화시키는 FSM은 위의 design에서 제시한대로 구현하였다.

아래는 각 state에서 control signal들의 활성화/비활성화 구현이다.

텍스트, 스크린샷, 폰트, 디자인이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

우선 기본적으로는 모든 control signal을 비활성화한다.

텍스트, 폰트, 스크린샷, 번호이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

IF state에서는 memory를 읽기 위해 mem\_read를 활성화하고, instruction register에 접근하기 위해 ir\_write를 활성화한다. ALU는 사용하지 않으므로 alu\_op에는 ALU\_NOP을 할당한다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

ID state에서는 ALU에서 PC+4를 계산해놓고 ALUOut 레지스터에 저장해놓는다. 또한 instruction이 ECALL일 경우, PC+4로 nextPC를 업데이트하기 위해 pc\_write을 활성화한다.

텍스트, 스크린샷, 폰트, 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

EX1 state는 R/I type, load, store, branch instruction일 때 ALU를 활성화한다. R type의 경우 ALU에서 레지스터 rs1, rs2의 연산 값을 ALUOut에 저장한다. I type은 레지스터 rs1과 immediate value의 연산 값을 ALUOut에 저장하고, Load와 Store는 rs1과 immediate value의 합을 ALUOut에 저장한다. Branch는 레지스터 rs1과 rs2에 저장된 값의 차를 ALU에서 계산하고, alu\_bcond를 결정한다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

EX2 state는 branch instruction일 때만 실행되며, EX1에서 결정된 alu\_bcond 값을 바탕으로 branch taken/not taken 상황을 구현한다. Branch가 taken되면 nextPC에 PC+Immediate value를 할당하고, not taken되면 nextPC에 PC+4를 할당한다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

MEM state는 instruction이 load/store일 때만 실행된다. Data memory에 접근하기 위해 i\_or\_d를 활성화하고, instruction이 load일 때는 메모리 값을 읽기 위해 mem\_read를 활성화한다. Instruction이 store일 때는 메모리 값에 write하기 위해 mem\_write를 활성화하고, PC+4로 PC를 업데이트한다.

텍스트, 스크린샷, 폰트, 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다. 텍스트, 스크린샷, 폰트, 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

WB state에서는 ALUOut 또는 메모리 값을 write back하기 위해 reg\_write를 활성화하고, PC가 업데이트되지 않은 instruction에서는 PC를 업데이트한다.

R/I type instruction에서는 ALUOut에 저장된 값을 rd 레지스터에 write back한다. 또한 PC를 PC+4로 업데이트한다. Load instruction에서는 메모리에서 읽은 값을 rd 레지스터에 write back하기 위해 reg\_write와 mem\_to\_reg를 활성화하고, 마찬가지로 PC를 PC+4로 업데이트한다. JAL instruction에서는 ALUOut에 저장되어 있던 PC+4를 rd 레지스터에 write하고, PC를 PC+Immediate value로 업데이트한다. JALR에서는 마찬가지로 ALUOut에 저장되어 있던 PC+4를 rd 레지스터에 write하고, PC를 GPR[rs1]+immediate\_value로 업데이트한다.

1. and\_gate, or\_gate, mux\_2x1, mux\_4x1

각각 CPU 상의 signal들을 and/or 연산으로 합쳐주거나, 회로 상 흐름을 조절하는 역할을 한다.

* clock asynchronous : clk이 들어오는 것과 관계없이 비동기식으로 input으로 받은

두 피연산자(mux의 경우 ctrl 신호도 input으로 필요)를 모듈의 목적에 맞게 연산 혹은 조건식 구성 후 계산이 완료된 result를 output으로 출력한다.

1. Discussion
2. Branch instruction - not taken에서의 next PC 구현

교안에 나타난 방식대로 branch instruction이 not taken되는 상황을 구현하고자 하였으나, cpu 모듈에서 alu\_bcond가 레지스터로 사용된다는 문제점이 있었다. 레지스터는 clock synchronous로, clk이 들어온 후 alu 연산을 바탕으로 alu\_bcond 값이 할당된다. 교안대로 구현한 경우 현재 instruction에서 레지스터 A, B의 연산 결과가 아닌, 이전 instruction에서 A, B에 저장되어 있던 값에 따른 alu\_bcond로 branch taken 여부가 결정되는 문제가 있었다. 이를 해결하기 위해 alu\_bcond가 0인 경우(branch not taken)도 EX2로 분리하여 구현하였다.

1. JAL 구현

JAL을 구현하며 ALUOut <= PC + 4 / RF[rd(IR)] <= ALUOut / PC <= PC+imm(IR)을 state를 나누어 구현하는 데 어려움을 겪었다. State를 여러 개로 쪼갰더니 레지스터에서 값이 사라지는 등의 문제가 있었고, State를 지나치게 통일해도 resource reuse 문제, signal 할당이 제대로 되지 않는 등의 문제가 있었다. 이를 해결하기 위해 JAL의 실행 흐름을 IF->ID->WB로 정리했으며, ID에서 ALUOut <= PC + 4, WB에서 RF[rd(IR)] <= ALUOut / PC <= PC+imm(IR) 를 수행하도록 구현하였다.

1. Conclusion

Lab3에서 작성한 코드로 주어진 테스트벤치를 실행한 결과는 아래와 같으며, 테스트벤치 실행 후 레지스터의 값들이 적절하게 할당된 것을 확인할 수 있다.

텍스트, 스크린샷이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.텍스트, 스크린샷이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.<basic, non-controlflow>

<if-else, loop, recursive>

텍스트, 스크린샷, 디자인이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.텍스트, 스크린샷이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.텍스트, 스크린샷, 디자인이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

Multi-cycle CPU는 하나의 cycle에 하나의 state가 수행되므로, instruction에 따라 소요되는 cycle이 달라진다. Lab3에서는 IF, ID, EX1, EX2, MEM, WB 총 6개의 state를 정의하여 구현하였는데, IF, ID등의 state 이름에 구애 받지 않고 기능에 초점을 맞춰 state를 정의한다면 소요되는 cycle 수를 더 줄일 수 있을 것이다.

Lab3를 통해 Multi-cycle CPU의 구조와 Multi-cycle CPU의 이점을 이해할 수 있게 되었다. 또한 디버깅을 진행하며 gtkWave를 잘 활용할 수 있게 되었다. 작성한 코드의 파형과 ripes에서의 register 값을 비교하며 달라지는 부분을 중점적으로 검토하니 어떤 instruction에서 틀렸는지, 어떤 signal이 할당이 안 되고 있는지 등을 추적하며 효과적으로 디버깅할 수 있었다.