Lab 4 : Four-stage Multicycle Processor Implementation

CSE 4190.308 Computer Architecture
2 Exercises (Total 70 Points)

Received: April 23, 2016
Due: 11:00 a.m., April 30, 2016

TA Office Hours: 7:00 - 8:00 p.m., 4/28 4/29

1 Introduction

This lab examines processor implementation in Bluespec. You will begin with a two-stage Y86-64 processor. For this lab, the first part of the lab requires you to implement a list of missing instructions and run a series of tests to validate the design. The second part of the lab requires you to divide the two-cycle implementation into a multi-stage non-pipelined version. This document describes the processor infrastructure, including how to build and run the processor to determine if it functions correctly and how well it performs, advice on how to debug the processor, the initial processor design, and detailed steps you should take to successfully implement the missing instructions.

1.1 Lab Organization

In the processor module used by your Lab 4 model, the file `Proc.bsv` describes the processor architecture. Figure 1 shows a rough sketch of the microarchitecture. There is a buffer register between the instruction fetch stage and Decode-Execute-Writeback stage. For this lab, all your code changes will be limited to two files: `Decode.bsv`, which implements the decode method used by the processor core, and `Proc.bsv`, two-stage multicycle processor.

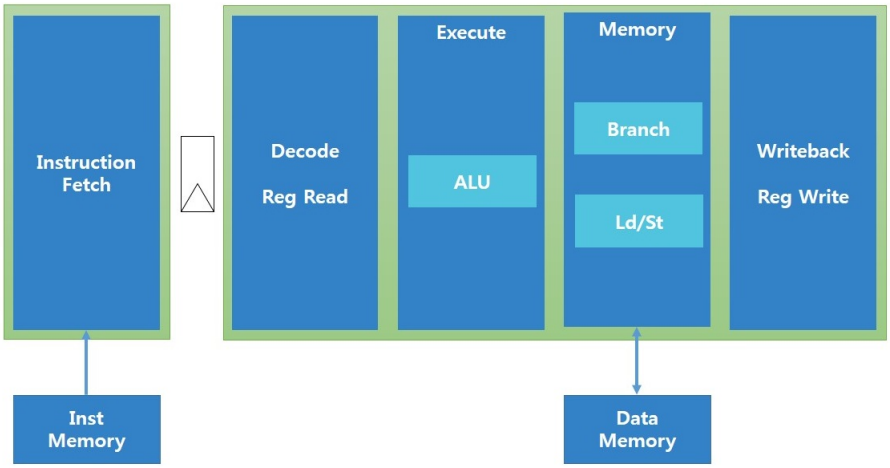

Figure 1: Two-stage Architecture Y86-64 Processor

The Y86-64 Instruction Set Specification can be found in the PDF file, `Chapter 5` available from the 'LECTURE NOTE' menu of the course webpage.

2 Getting Started

(If you need a language assistance, please see our English-speaking TAs.)

2.1 How to Download the Source Code

컴퓨터 구조 과목에서 사용하게 될 모든 실습은 이전에도 언급되었듯이 svn 서버를 통해 관리됩니다. Lab 4의 실습 코드를 받는 방법은 기존 Lab들과 동일합니다. 수업 홈페이지에 올라와 있는 `add-lab4.sh` 스크립트를 다운받고, 이전 Lab을 수행했던 실습 디렉터리의 상위에 위치시킵니다.

```
$ ls
add-lab4.sh YOUR_ID/
$ ls YOUR_ID/
lab0/ lab1/ lab2/ lab3/
```

그 후 아래 예제와 같이 본인의 ID를 인자로 스크립트를 실행하셔야 합니다. 이 아이디는 첫 실습 과제에 앞서 `student-create.sh` 스크립트로 생성했던 아이디입니다. 코드를 다운 받는 과정에서 archi16의 password를 요구할 경우 강의 홈페이지와 동일한 비밀번호를 입력하시면 됩니다.

다음으로, svn 계정의 비밀번호를 입력해야 이 과정을 무사히 마칠 수 있습니다. svn 계정의 아이디는 `student-create.sh` 스크립트로 생성했던 아이디이며, 비밀번호는 공지한 바와 같이 조교에게 이메일로 보내주신 희망 비밀번호로 설정되어 있습니다.

```
$ ./add-lab4.sh YOUR_ID
Getting source codes for YOUR_ID
archi16@hyewon.snu.ac.kr's password:
...
Checking in initial repository
Authentication realm: <svn://hyewon.snu.ac.kr:3690> 2016 Computer Architecture
Password for 'YOUR_ID':
...
Transmitting file data .....
Committed revision 14.
```

위와 같이 새로운 revision 번호를 알려주는 메세지가 보이면 실습 코드 다운이 완료된 것입니다. 실습 디렉터리 아래에 `lab4` 디렉터리가 새롭게 생성되었을 것입니다.

```
$ ls YOUR_ID/
lab0/ lab1/ lab2/ lab3/ lab4/
```

2.2 Directory Structure of Lab 4

Lab 4 실습의 디렉터리 구조는 다음과 같습니다.

```
lab4/
    build/
    lib/
       common-lib/
                  ProcTypes.bsv
    src/
       Decode.bsv
       Proc.bsv
       decode_result.orig
       y86
```

`build/`
: 컴파일 시 생성되는 파일들이 위치하는 폴더입니다.

`lib/`
: 실습에 사용되는 library 파일들이 위치하는 폴더입니다. 실습을 하면서 수정할 필요 없는 Bluespec 소스 파일들을 포함하고 있습니다.

`lib/common-lib/ProcTypes.bsv`
: Y86-64 프로세서의 타입을 지정해둔 파일입니다. 이 파일을 참조하여 앞으로 프로세서 관련 Lab에서 쓰일 Type들의 정의를 볼 수 있습니다.

`src/Decode.bsv`
: Y86-64 프로세서에서 80bit 명령어를 Decoding하는 것에 대한 모듈입니다. Exercise 1에서 고치게 될 파일입니다.

`src/Proc.bsv`
: 위에서 구현한 두 모듈을 import하여 프로세서를 구현하는 파일입니다. Exercise 2에서 고치게 될 파일입니다.

`src/decode_result.orig`
: 디코딩 결과의 모범답안 입니다. 구현한 디코딩 로직의 검증 결과로 생성되는 `decode_result`와 diff명령을 통하여 비교를 하면 로직을 정상적으로 구현하였는지 확인할 수 있습니다.

`src/y86`
: Exercise 1, 2에서 구현할 프로세서에 대해 컴파일 및 검증을 할 수 있도록하는 스크립트 파일입니다.

2.3 How to Simulate the Design

Lab 4는 크게 디코딩 로직의 구현과 two-stage multi-cycle 프로세서를 four-stage multi-cycle 프로세서로 변환하는 과제로 나눠집니다.

2.3.1 Compiling and Simultation in Exercise

구현할 디코딩 로직과 four-stage multi-cycle 검증은 아래와 같이 `y86` 스크립트를 실행하여 이뤄집니다.

```
$ ./y86 -c [-d]
$ ./y86 -r [-d]
```

먼저, -c 플래그로 구현한 Y86-64프로세서를 컴파일을 할 수 있습니다. 다음, -r 플래그로 모든 명령어 수행을 테스트하거나, `-o` 플래그를 통해 특정 명령어를 테스트할 수 있습니다. 각 명령어 테스트가 끝나면, PASS 여부 및 수행한 사이클과 명령어 수가 출력됩니다.
-d 플래그를 추가로 주면 디코드 테스트를 위한 컴파일과 명령어 수행을 할 수 있습니다. 테스트 디코딩 수행 결과는 `src`폴더내에 `decode_result`라는 텍스트 파일이 저장되고, 이 파일을 제공되는 올바른 결과인 `decode_result.orig` 파일과 비교하여 디코딩 로직이 제대로 구현됐는지 확인 할 수 있습니다. 참고로, Linux에서 아래와 같은 명령어를 이용하면, 생성되는 `comp`파일에서 두 파일의 다른 점을 비교할 수 있습니다.

```
$ diff decode_result decode_result.orig > comp
```

3 Implementing Decoding Module

`lab4/src/Decode.bsv` 의 decode 함수는 80bit의 명령어를 정해진 형식에 따라 디코딩하는 함수입 니다. 디코딩 결과는 `ProcType.bsv`에 정의돼있는 `DecodedInst`와 같습니다. valA, valB, copVal 필드는 디코딩 이후 사용되는 필드로 Decode에서 설정하지 않습니다.

```
typedef struct{
        IType   iType;
        OpqFunc opqFunc;
        CondUsed  condUsed;
        Addr valP;
        Maybe#(FullIndx) dstE;
        Maybe#(FullIndx) dstM;
        Maybe#(FullIndx) regA;
        Maybe#(FullIndx) regB;
        Maybe#(Data) valA;
        Maybe#(Data) valB;
        Maybe#(Data) valC;
        Maybe#(Data) copVal;
} DecodedInst deriving(Bits,Eq);
```

IType iType
명령어의 타입을 나타냅니다. 마찬가지로 `ProcType.bsv`에 정의 되어 있습니다. `Mfc0, Mtc0`에 대해서는 이미 구현이 되어있으므로 이 과제에서는 무시해도 무방합니다.

```
typedef enum{
        Unsupported,
        Rmov,
        Opq,
        RMmov,
        MRmov,
        Cmov,
        Jmp,
        Push,
        Pop,
        Call,
        Ret,
        Hlt,
        Nop,
        Mtc0,
        Mfc0
}IType deriving(Bits, Eq);
```

OplFunc oplFunc
디코딩 되는 명령어가 어떤 종류의 Opq 연산을 사용하는지를 나타냅니다. 마찬가지로 `ProcType.bsv`에 정의 되어 있습니다.

```
typedef enum{
        FNop,
        FAdd,
        FSub,
        FAnd,
        FXor
} OplFunc deriving(Bits, Eq);
```

CondUsed condUsed
디코딩 되는 명령어가 어떤 종류의 Cond 연산을 사용하는지를 나타냅니다. 마찬가지로 `ProcType.bsv`에 정의 되어 있습 니다.

```
        typedef enum{
                Al,   // Unconditional
                Eq,   // Equal
                Neq,  // Not Equal
                Lt,   // Less than
                Le,   // Less than or Equal
                Gt,   // Greater than
                Ge    // Greater than or Equal
        } CondUsed deriving(Bits, Eq);
```

Addr valP
다음 실행할 명령어 주소를 나타냅니다.

Maybe#(Fullindx) dstE
Destination 레지스터의 번호를 나타냅니다.

Maybe#(Fullindx) dstM
메모리에서 값이 로드될 Destination 레지스터 번호를 나타냅니다.

Maybe#(Fullindx) regA
첫 번째 Source 레지스터의 번호를 나타냅니다.

Maybe#(Fullindx) regB
두 번째 Source 레지스터의 번호를 나타냅니다.

Maybe#(Data) valC
명령이 immediate 값을 사용할 경우, 사용할 immediate 값을 나타냅니다.

디코딩을 올바르게 수행하기 위하여, `decode`함수 안의 `case` 문에 있는 경우마다 정확한 값을 위에 설명한 필드마다 채워 넣어야 합니다. 경우마다 디코딩된 명령이 참고할 필드만 신경써서 채우면 되고, 꼭 모든 필드 마다 새로운 값을 넣을 필요는 없습니다.
예를 들어 `mrmovq`와 같은 명령어 는 `valC` 값을 사용하지만, `cmov`와 같은 명령어는 `valC` 값을 사용하지 않습니다. 따라서 `cmov`와 같은 경우, `valC`는 `Invalid`로 설정하지만, `mrmovq`의 경우에는 `valC`의 필드에 정확한 값을 설정해주어야 합니다. 이와 같이 각 명령 마다 `DecodedInst`의 필드가 어떻게 사용되는지를 확실하게 숙지하고, 필요한 필드를 새로 설정하여야 합니다.

수정해야할 `decode` 함수의 구조는 아래와 같습니다.

```
function DecodedInst decode(Inst inst, Addr pc);
        DecodedInst dInst = ?;
        let iCode = inst[79:76];
        let ifun  = inst[75:72];
        let rA    = inst[71:68];
        let rB    = inst[67:64];
        let imm   = little2BigEndian(inst[63:0]);
        let dest  = little2BigEndian(inst[71:8]);

        case (iCode)
        halt, nop :
        begin
                ...
        end
        irmovq :
```

```
        begin
                ...
        end
        ...
        endcase
        return dInst;
endfunction
```

리턴 값인 dInst의 초기화 이후에, inst 인자를 bit concatenate 함수를 이용하여 필요한 값들을 설정해놓았습니다. 따라서 나머지 디코딩 모듈 구현을 위해서는 inst는 다시 참조할 필요가 없고, 새로 설정된 값들이 의미하는 바가 무엇인지를 이해하고 새로운 리턴 값을 만드는데 사용하면 됩니다.

case 문은 iCode 에 따라 dInst의 필드를 재설정해주는 방식으로 되어있습니다. 올바른 디코딩 모듈의 구현을 위해, 명령어에 따라 설정해야할 필드가 무엇인지를 이해하여 모든 명령어에 대해 빠짐없이 정확한 필드값을 설정해야 합니다.

Exercise [1] (30 points): Decode.bsv 파일 안의 decode함수를 완성하시오. 다음 Exercise로 넘어가기 전에 반드시 y86 스크립트를 통해 결과를 확인할 것.

4 Multi-cycle Y86-64 implementation

The provided code implementes a two-cycle non-pipelined Y86-64 implementation. Your goal for this lab is to split one of the stages into three stages, making the processor a four-cycle non-pipelined implementation.

Inside the provided Proc.bsv file, you can find the module mkProc, which implements two rules doFetch and doExecute. These rules are scheduled one after another by setting the register stage of type Stage. Your job is to split the doExecute rule into three rules: doExecute, doMemory and doWriteBack. You can add these stages into the Stage type by adding them to the typedef enum statement.

The doExecute stage should take care of decode and execute. The doMemory stage should take care of the dMem requests, and doWriteBack stage should take care of the writeback to rf.

In order to reduce the cycle count, another requirement is to skip the doMemory stage whenever it is not needed. So when there is no memory access instructions, the processor should behave as a three-stage non-pipelined processor.

Exercise [2] (40 points): Divide the doExecute rule into doExecute, doMemory and doWriteBack, to make the processor a four-cycle non-pipelined implementation. The doMemory rule should be skipped whenever it is not required. You can check the impact of your modifications through the script output.