Lab 6: Three Stage Processor Implementation and Branch Prediction

CSE 4190.308 Computer Architecture
3 Exercises (Total 100 Points)
Recieved: May 8, 2012
Due: 9:10 a.m., May 22, 2012

1 Introduction

Lab 6에서는 Lab 4, 5에 걸쳐 구현하였던 2-stage pipeline 프로세서를 확장하여 3-stage pipeline 프로세서를 구현합니다. 아울러 stage의 수가 증가함으로 인해 나타나게되는 data hazard를 효과적으로 해결합니다. Data가 stage 간에 의존성이 생길 때, 단순히 stage의 동작을 stall하는 것이 아닌, bypass 가능한 pipeline을 설계합니다. 이로써 stage를 세분화 하여 clock frequency를 늘리면서도, 프로세서의 IPC를 (Instructions Per Clock) 유지할 수 있습니다. 또한, fetch 단계에서 다음 cycle에서 fetch해야할 명령어의 pc를 미리 예측하는 branch prediction module을 구현합니다. 이를 통해 branch miss로 인한 stall 횟수를 최소화 함으로써 더 높은 성능의 프로세서를 구현할 수 있습니다.

1.1 Lab Organization

Lab 6에서 구현할 프로세서의 구조는 Figure 1과 같습니다. `ThreeStageHavard.bsv`은 Lab 6에서 구현할 프로세서를 기술하는 bluespec 파일입니다. Lab 5에서 pipeline을 `IF-DEC` (Instruction Fatch, Decoding), `EXEC-MEM-WB` (Execution, Memory, Writeback)와 같이 나눈 것을 확장하여, Lab 6에서는 `IF-DEC`, `EXEC-MEM`, `WB`의 세 단계로 stage를 나누게 됩니다. `IF-DEC` stage 와 `EXEC-MEM` stage가 나누어져 있기 때문에 Lab 5와 마찬가지로 control hazard가 존재하며, 레지스터를 읽어오는 동작은 `IF-DEC` stage에서, 레지스터의 값을 변하게 하는 동작은 `EXEC-MEM, WB` stage 에서로 분리되기 때문에 data hazard 또한 고려해야 합니다.

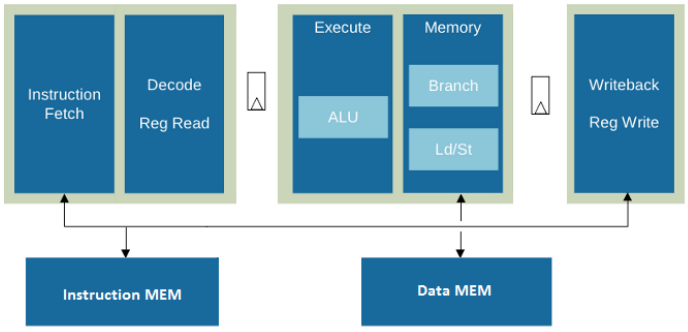

Figure 1: Three-stage Havard Architecture SMIPSv2 Processor

이와 같이 2-stage pipeline에서 3-stage pipeline으로 확장하게 되면, stage의 길이를 줄임으로써 clock period를 줄일 수 있지만, 기존에 존재하지 않았던 문제들(이 경우에는 data hazard)로 인하여 성능이 감소할 수 있습니다. 실제로, bypass logic이 없는 3-stage pipeline에서 data hazard를 피하기 위해 `IF-DEC` stage를 stall하는 경우, IPC가 Lab 4, 5에서보다 크게 증가하게 됩니다. 따라서 프로세서의 성능을 기존 보다 실질적으로 향상 시키기 위해서는 data hazard를 효과적으로 처리할 수 있어야 합니다.

2 Getting Started

2.1 How to Download the Source Code

이전 Lab들과 마찬가지로, 본 문서와 같은 곳에 업로드 되는 `add-lab6.sh` 스크립트를 다운로드 받아 실행하여 Lab 6의 실습 코드를 받을 수 있습니다.

```
$add-lab6.sh (student id)
```

2.2 Directory Structure of Lab 6

Lab 6 실습의 디렉토리 구조는 Figure 2와 같습니다.

```
lab6/
    build/
        bdir/
        simdir/
        data/
            smips/
            benchmarks/
    lib/
        common/
        memory/
        processor/
            ProcTypes.bsv
            EHRPipeReg.bsv
    src/
        ThreeStageHavard.bsv
        Btb.bsv
        test_script.sh
        benchmark_script.sh
```

Figure 2: Lab 6 코드 디렉토리 구조

build/
컴파일 시 생성되는 파일들이 위치하는 폴더입니다.

build/data/smips
시뮬레이션에 사용할 명령어 별 어셈블리 파일들을 가지고 있습니다.

build/data/benchmarks
시뮬레이션에 사용할 벤치마크들의 어셈블리 파일들을 가지고 있습니다.

lib/
Lab 6에서 쓰일 Bluespec 라이브러리들을 가지고 있습니다.

lib/processor/ProcTypes.bsv
SMIPS 프로세서의 타입을 지정해둔 파일입니다. 프로세서 구현에 쓰이는 data type들과 함수, 인터페이스의 정의를 볼 수 있습니다.

lib/processor/EHRPipeReg.bsv
Pipeline 레지스터를 정의한 파일입니다. Bypassing을 구현하기 위해서 pipeline 레지스터의 인터페이스와 모듈 구현을 수정해야 합니다.

src/ThreeStageHavard.bsv
SMIPS 프로세서 모듈을 기술한 파일입니다. 이 파일을 수정하여 Lab 6를 진행합니다.

src/Btb.bsv
Branch prediction 모듈을 정의하는 파일입니다. 단순히 'pc + 4' 로 branch prediction을 하는 초기 모듈을, branch target buffer를 활용하는 방식으로 수정하도록 합니다.

src/test_script.sh
SMIPSv2 파일들을 시뮬레이션의 입력으로 설정하는 스크립트 파일입니다.

src/benchmark_script.sh
벤치마크 파일들을 시뮬레이션의 입력으로 설정하는 스크립트 파일입니다.

2.3 How to Simulate Your Design

기존의 Lab 4, 5와 마찬가지로 Bluespec workstation에서가 아닌 command line에서 직접 명령을 실행하여 컴파일 및 시뮬레이션을 수행합니다. 먼저 모듈의 컴파일은

```
$./make
```

를 통하여 이뤄지게 됩니다. 프로세서 동작의 확인 과정 역시 Lab 4에서 설명한 내용과 동일합니다. 어셈블리 코드 파일들을 이용하는 시뮬레이션은

```
$./test_script.sh (instruction)
$./test_script.sh all
```

와 같은 명령을 통해 수행할 수 있습니다. (instruction) 부분에는 addu, or, sw, beq 등과 같은 SMIPSv2의 명령어들이 들어갈 수 있습니다 (Lab 4 문서 및 /build/data/smips 폴더 참조). 수 행 후 화면에 PASSED 문구가 나타나면 해당 어셈블리 코드는 프로세서 내에서 정상 동작하였음을 의미합니다. 반대로 FAILED나 It's been a long time running... 문구가 나타나면 구현한 프로 세서가 해당 명령어들을 정상적으로 수행하지 못했음을 나타냅니다. 벤치마크 코드 파일을 이용하는 시뮬레이션은

```
$./benchmark_script.sh (benchmark)
$./benchmark_script.sh all
```

와 같은 명령을 통해 수행할 수 있습니다. (benchmark) 부분에는 median, multiply, qsort, towers, vvadd 등과 같은 benchmark 코드 이름들이 들어갈 수 있습니다 (/build/data/benchmarks 폴더 참조). 역시 정상적으로 수행이 완료되면 PASSED 문구가 표시될 것입니다. 벤치마크 파일을 이용하는 시뮬레이션의 경우에는 추가적으로 수행한 명령어의 수 및 수행에 걸린 cycle의 수를 같이 표시합니다. 두 경우 모두, 함께 생성되는 out.tex 파일을 참고하면 시뮬레이션이 실행된 자세한 과정을 볼 수 있습니다.

2.4 How to Submit Your Design

이전 Lab들에서와 같이 lab6 폴더에서 svn commit 명령을 수행하면 수정된 파일들이 제출됩니다.

3 Implementing Three-Stage Pipeline

Lab 6에서 쓰일 프로세서를 기술하는 lab6/src/ThreeStageHavard.bsv파일을 보면, doFetch, doExcute, doCommit와 같은 세 개의 룰로 stage가 나누어져 있는 것을 볼 수 있습니다. 각 룰은 앞에서 언급한 바와 같이 IF-DEC, EXEC-MEM, WB 단계의 일을 하게 됩니다. 초기 파일에는 각 룰이 비어있으므로, 기존 Lab의 코드와 수업에서 사용한 자료를 참고하여 각 stage에 알맞는 동작을 채워넣어야 합니다. 이를 위해 필요한 data type은 Figure 3과 같이 ThreeStageHavard.bsv에 정의돼있는 mkProc 모듈 상단에 선언 되어 있습니다.

Figure 3에서 주석 아래에 존재하는 변수들은 Host와 CPU간의 통신과 관련된 것으로, Lab 6의 구현에서는 고려하지 않아도 좋습니다(잘못 수정할 경우, 의도치 않은 시뮬레이션 결과가 발생할 수 있음). 따라서, 주석보다 상단에 있는 변수들을 사용하여 프로세서를 구현하도록 합니다. 보다 자세한 data type들의 정의는 /lab6/lib/processor/ProcType.bsv를 참고합니다. 본 문서에서는 Lab 6에서 새로 구현된 EHR을 이용한 pipeline 레지스터에 대해 알아보겠습니다. EHR을 이용한

각종 레지스터들의 인터페이스와 모듈 구현은 /lab6/lib/common/EHRPipeReg.bsv에서 참고 할 수 있습니다. 그 중 Lab 6에서 pipeline 레지스터로서 사용하는 EHRSPipieReg의 인터페이스는 Figure 4와 같습니다.

```
module mkProc(Proc);
    Reg#(Addr) pc    <- mkRegU;
    RFile      rf    <- mkEHRBypassRFile;
    IMemory    iMem  <- mkIMemory;
    DMemory    dMem  <- mkDMemory;

    Reg#(Bool) fEpoch    <- mkReg(False);
    Reg#(Bool) eEpoch <- mkReg(False);

    EHRSPipeReg#(TypeDecode2Execute) itr <- mkEHRSPipeReg(getDstE);
    EHRSPipeReg#(TypeExecute2Commit) cr  <- mkEHRSPipeReg(getDstC);

    EHRBypassReg#(Addr) nextPC <- mkEHRBypassReg;

    //For Host-CPU communication. Don't touch these.
    EHR#(2, Data)    cp0_tohost <- mkEHR(0);
    ...
```

Figure 3: 프로세서 모듈에서 쓰일 data type의 선언

```
interface EHRSPipeReg#(type dataT);
    method Action enq(dataT d);
    method Action deq;
    method Action clear;
    method dataT  first;
    method Bool   notEmpty;
    method Bool   notFull;
    method ActionValue#(Bool) search(Maybe#(Rindx) s1, Maybe#(Rindx) s2);
endinterface
```

Figure 4: EHRSPipeReg의 인터페이스

ESRSPipeReg 인터페이스를 보면, Lab 5에서 사용한 파이프라인 프로세서와 큰 차이는 없지만, 새로운 메소드인 search가 추가된 것을 알 수 있습니다. search는 레지스터의 인덱스를 의미하는 Rindx type 인자 두 개(fetch된 명령에는 두 개의 소스 레지스터가 존재할 수 있으므로)를 받아 그것들이 현재 파이프라인 레지스터에 저장돼있는 명령어에서 destination 레지스터의 인덱스와 같은지를 Bool type으로 리턴하게 됩니다. 메소드의 구현은 Figure 5와 같이 이루어 집니다.

모듈 이름인 mkEHRSPipeReg 옆에 #과 함께 들어가는 괄호에는, 모듈을 만들 때 필요한 인자를 입력합니다. 지금까지 사용해온 레지스터 모듈을 예로 들면, mkReg#(0)과 같이 mkReg라는 모듈 이름 옆에 초기값을 넣어 주었던 것과 같습니다. EHRSPipeReg type은 Figure 3에서 선언한 것과 같이 다른 종류의 명령어 타입(TypeExecute2Commit, TypeExecute2Commit)을 저장할 수 있습니다.따라서, 해당하는 명령어 타입에 따라 destination 레지스터의 인덱스를 찾는 방법이 달라질 수 있습니다. 그러한 이유로, 모듈을 선언할 때 저장되는 명령어 타입에 따라 destination 레지스터의 인덱스를 찾는 함수를 인자로 넣어 줌으로써 모듈을 polymorphism하게 구현할 수 있습니다. Figure 3에서 EHRSPipeReg 선언시 인자로 넣어준 getDstE, getDstC 함수는 TypeExecute2Commit, TypeExecute2Commit과 같은 각각의 type에서 destination 레지스터(Rindx type)를 구해주는 함수입니다. 이렇게 구한 destination 레지스터 인덱스와 인자로 받는 레지스터의 인덱스들을 dataHazard

```
module mkEHRSPipeReg#(function Maybe#(Rindx) getDst(dataT d)) (EHRSPipeReg#(dataT));
    ...
    method notFull = pQ.notEmpty;
    method ActionValue#(Bool) search(Maybe#(Rindx) s1, Maybe#(Rindx) s2);
        if(sQ.notEmpty) begin
            sQ.deq;
            return dataHazard(s1, s2, getDst(sQ.first));
        end
        else begin
            return False;
        end
    endmethod
endmodule
```

Figure 5: mkEHRSPipeReg의 구현

함수에 넣으면, s1, s2와 destination 레지스터의 인덱스가 같은 지를 알 수 있습니다. 따라서 Figure 6와 같은 방법으로 Bool 값을 받아 각각의 stage에서 data의 의존성이 생기는지를 확인 할 수 있습니다.

```
let stallE = itr.search(rSrc1, rSrc2);
let stallC = cr.search(rSrc1, rSrc2);
// rSrc#: the register index of the source register #
```

Figure 6: search메소드의 사용예

./benchmark_script.sh 스크립트를 실행시켜 완성된 프로세서의 동작을 확인할 수 있습니다. 시뮬레이션 결과로 나오는 처리한 명령어의 수와 소모되는 cycle 수를 이용하여 IPC를 계산할 수 있습니다. EXEC-MEM stage와 WB stage에서 데이터 의존성이 생길 때 마다 IF-DEC stage를 stall할 경우, IPC 계산 결과는 다음과 같이 나오게 됩니다.

Benchmark	Number of instuctions	Clock cycles	IPC
median	5671	10457	0.542
multiply	21194	40621	0.522
qsort	21622	38967	0.555
towers	9903	17086	0.580
vvadd	3012	4521	0.666

Table 1: Stall을 통해 data hazard를 해결한 프로세서의 IPC

Exercise [1] (20 points): Lab 4, 5의 코드와 수업의 강의자료를 참고하여 3-stage pipeline 프로세서를 완성하시오. 데이터의 의존성이 있을 때는 IF-DEC stage:doFetch 룰이 stall되도록 구현하시오. doFetch 룰에서는 명령어의 fetching, decoding, 레지스터 read가 일어나고, doExcute 룰에서는 명령의 execution과 memory 관련 연산이 일어나며, doCommit 를에서는 레지스터 파일에 결과를 저장하도록 구현하시오.

4 Handling Data Hazard With Bypassing

Introduction에서 언급한 바와 같이, pipeline stage를 늘림으로써 clock frequency를 증가시킬 수 있지만, 그에 따라 새로운 문제가 발생할 수 있습니다. Lab 6의 경우에는 발생하는 data hazard로

인한 stall을 효과적으로 다루지 못하면 결국 IPC가 감소하게 되므로 프로세서의 실질적인 성능 향상을 이루기 어려워집니다. Lab 6의 두 번째 목표는 `IF-DEC` stage에서 레지스터 값을 읽어올 때, 의존성이 존재하는 data를 bypass하게 전해주도록 하는 logic을 구현하는 것입니다. Bypass logic을 통해 data hazard로 인한 `IF-DEC` stage의 stall을 없앨 수 있고, 결과적으로는 한 clock 사이클에서 모든 stage가 동작할 수 있는 clock을 늘리기 때문에 IPC를 1에 가깝게 끌어올릴 수 있습니다.

이를 Exercise 1에서 완성한 프로세서에서 구현하기 위해서는 pipeline 레지스터로서 사용했던 `EHRSPipeReg`를 수정하거나 그것을 대체할 새로운 type을 만들 필요가 있습니다. `EHRSPipeReg`는 입력으로 받는 source 레지스터들의 인덱스가, 저장된 명령어에서 destination 레지스터의 인덱스와 겹치는지만을 판단하였기 때문에, 두 레지스터 중 어떤 레지스터와 겹치는지, 혹은 저장된 명령어가 실제로 레지스터를 업데이트 할 것인지, 그리고 전달이 필요한 실제 레지스터에 업데이트 되는 값을 알 수 없습니다. 따라서 data hazard를 bypassing을 통해 해결하려면,

1. 레지스터 read 시 source 레지스터의 인덱스와 다른 stage에서의 destination 레지스터의 인덱스가 겹치는지
2. 겹치는 레지스터의 인덱스가 두 source 레지스터 중 어느 것인지
3. `EXEC-MEM, WB`에서 수행하는 명령어가 실제로 레지스터를 업데이트 하는지
4. 업데이트가 일어날 때 destination 레지스터에 쓰일 값이 무엇인지

를 `IF-DEC` stage에서 알 수 있도록 pipeline 레지스터의 인터페이스(메소드)를 수정하거나 새로운 bypass 레지스터를 추가해야 합니다. `IF-DEC` stage에서 위의 네 가지 정보를 받아 source 레지스터에 적절한 값을 정하여(레지스터 파일에서 읽거나, 다른 stage로 부터 bypass logic을 통해 받아) 다음 stage로 넘겨줌으로써 data hazard로 인한 stall이 일어나지 않게하고, 결과적으로 프로세서의 clock frequency와 IPC를 동시에 올릴 수 있습니다.

Exercise 1과 마찬가지로 `./benchmark_script` 스크립트를 실행시켜 완성된 프로세서의 동작을 확인할 수 있습니다. Bypassing을 통해 data hazard를 관리하는 프로세서를 구현했을 때, IPC 계산 결과는 다음과 같이 나오게 됩니다.

Benchmark	Number of instuctions	Clock cycles	IPC
median	5671	7021	0.808
multiply	21194	27311	0.776
qsort	21622	24523	0.882
towers	9903	10985	0.902
vvadd	3012	3315	0.909

Table 2: Bypass 통해 data hazard를 해결한 프로세서의 IPC

Exercise [2] (30 points): Exercise 1에서 구현한 프로세서에서 data hazard를 bypassing을 통해 관리하도록 구현하시오.

5 Implementing A Branch Prediction Module

Lab 6의 마지막 목표는 branch prediction module을 구현하여 control harzard로 인한 stall 횟수를 줄이고, 이를 통해 IPC를 1에 가깝게 만들어 프로세서의 성능을 개선하는 것입니다. Lab 4, 5에서 구현한 프로세서에서는 branch prediction을 항상 `'pc + 4'` 로 간단히 해왔습니다. 즉, 항상 branch가 taken 되지 않을 것이라고 예측(prediction)했기 때문에 다음 pc에 해당하는 명령을 fetch하도록 했습니다. 이 예측이 틀렸을 경우(miss prediction), 즉, branch가 taken 되었을 경우 control hazard를 피하기 위해 epoch을 바꾸었으며, stall되는 횟수를 최소화 하도록 bypass 레지스터를 사용하여 다음 명령어를 읽어올 pc값을 `IF-DEC` stage로 전달하였습니다. Lab 6에서는 이러한 기초적인 예측을 개선하여 miss prediction의 횟수를 줄일 수 있도록 합니다. 이를 위하여 BTB (Branch Table Buffer)방식의 branch prediction module을 구현하도록 합니다.

5.1 Branch Prediction with BTB: Branch Table Buffer

BTB를 이용한 branch prediction의 알고리즘은 아래 psuedo 코드와 같습니다.

```
if ( There is a (pc, next pc) entry in BTB ) then
    predicted pc = next pc;
else
    predicted pc = pc + 4;
```

BTB에는 어떤 pc에 대해 그 다음에 fetch해야할 pc의 순서쌍을 entry로 삼아 기록합니다. BTB의 초기상태는 아무런 entry가 없으므로, 처음에는 모든 pc에 대해 `'pc + 4'`를 다음에 fetch해야할 pc로 예측하게 됩니다. BTB는 miss prediction이 일어났을 때 업데이트 되며, miss prediction을 발생시킨 pc와 fetch 했어야 할 pc를 순서쌍으로 삼아 BTB에 저장하게 됩니다. 즉, 어떤 pc 이후에 fetch 됐어야 할 pc가 기존의 prediction에 의한 (초기 prediction은 `'pc + 4'`) pc 값과 달라질 경우, 다음부터의 예측은 그 결과를 이용하게 하는 방법입니다. 이 방법은 매우 간단하지만, for-loop condition에 따른 branch와 같은 경우, 기초적인 `'pc + 4'`를 통한 예측보다 miss prediction을 크게 줄일 수 있습니다. 예를 들어,

```
for(i = 0 ; i < 16 ; i++)
{
    ...
}
```

과 같은 for-loop에서, `'pc + 4'`로 예측할 경우 총 열 다섯 번의 miss predition이 발생합니다. 하지만 BTB를 이용한 predictiond의 경우, 초기 한 번의 miss prediction 이후에 loop-condition을 확인하는 branch에서의 next pc값을 loop의 처음 명령어에 대한 pc로 설정합니다. 이렇게 하면 loop을 탈출하기 전까지 열 네 번의 prediction은 맞는 예측이 됩니다. 따라서 열 다섯 번의 misprediction은 loop을 탈출할 때 한 번을 포함하여 총 두 번으로 줄어듭니다. Branch의 발생 횟수 중 많은 부분을 loop-structure가 차지할 것이라 가정하면, 전체적인 성능을 기초적인 prediction에 비해 크게 향상시킬 수 있습니다.

5.2 Interface of Branch Prediction Module

Lab 6에서 구현하게 될 branch prediction module의 인터페이스 `lab6/lib/common/Btb.bsv` 파일에 정의돼있으며, Figure 7과 같습니다.

```
interface NextAddressPredictor;
    method Addr prediction(Addr pc);
    method Action update(Addr pc, Addr target);
endinterface
```

Figure 7: BTB를 이용한 branch prediction module의 인터페이스

Lab 6에서 이와 같은 branch prediction module을 활용하기 위해서는 해당 모듈을 프로세서 모듈(`mkProc`)에서 선언하고, pc와 관련된 변수를 다루는 부분을 수정해 주어야 합니다. 우선, branch prediction module을 따로 사용하지 않는 경우, miss prediction은 branch가 taken 됐을 경우를 뜻하므로, `EXEC-MEM` stage에서

```
let eInst = exec(dInst, rVal1, rVal2, itrpc, ?, ?);
...

if(eInst.brTaken) begin
    nextPC.enq(eInst.addr);
    eEpoch <= !eEpoch;
end
```

와 같이 miss prediction 여부를 파악하였고, 그 경우 bypass 레지스터인 nextPC를 통해 제대로 된 pc을 IF-DEC stage로 전달하였습니다. 이와 달리 BTB를 사용한 경우, 초기에 branch가 taken되어 'pc + 4'에 의한 miss prediction이 발생하게 되면 실제 예측했어야 할 pc값을 BTB에 저장하고, 다음 branch에서는 저장한 pc 순서쌍을 이용하게됩니다. 따라서 miss prediction의 판단을 단순히 branch의 taken 여부만으로 판단할 수 없습니다. 그 결과, branch prediction moduled을 사용하기 위해서는 miss predictiond의 여부를 판단하는 다른 방법이 필요하게 됩니다. Lab 6에는 EXEC-MEM stage에서 exec 함수를 이용하여 miss prediction 여부를 검사하고(이미 logic이 구현되어 있음), 그 결과를 return 값(ExecInst type)의 Bool misprediction 필드에 저장하게 됩니다. 따라서 miss prediction의 여부는

```
let eInst = exec(dInst, rVal1, rVal2, itrpc, itrppc, ?);
...

if(eInst.misprediction) begin
    nextPC.enq(...);
    eEpoch <= !eEpoch;
end
```

과 같이 판단할 수 있습니다. branch prediction module과 비교해보면 exec 함수에서 itrpc 값 옆의 입력값이 ?에서 itrppc로 변한 것을 확인 할 수 있습니다. 이는 exec 함수에서 miss prediction 여부를 판단하기 위해 예측한 pc (predicted pc)와의 비교가 필요하므로 BTB를 쓰기전에는 "Don't care"(?)였던 ppc 필드의 입력값을 itrppc로 명시한 것입니다. 마찬가지로, IF-DEC stage에서 pipeline 레지스터를 통해 decode된 명령어를 전달할 때

```
itr.enq(TypeDecode2Execute{pc:pc, epoch:fEpoch, inst:inst, ...});
```

와 같이 예측한 pc를 따로 전달하지 않았으나(필드는 TypeExecute2Commit에 미리 정의 돼있음), BTB를 사용하기 위해서는 다음과 같이 ppc 필드에 branch prediction module로 부터 얻은 값을 명시하여 EXEC-MEM stage로 전달해야 합니다.

```
module mkProc(Proc);
    ...
    NextAddressPredictor bpred <- mkBTB; // declaration of branch predictor
    ...
    doFetch(cp0_fromhost && itr.notFull);
        ...
        let ppc = bpred.prediction(pc); //get predicted pc
        ...
        itr.enq(TypeDecode2Execute{pc:pc, ppc:ppc, epoch:fEpoch, inst:inst, ...});
        ...
```

또한 bypass 레지스터인 nextPC의 경우, BTB의 업데이트를 위해서는 EXEC-MEM stage에서 제대로 fatch했어야 할 next pc 뿐만 아니라, 어떤 pc가 misprediction을 발생했는지도 IF-MEM stage에 전달해주어야 합니다. 따라서 nextPC 레지스터가 저장해야할 type이 달라져야 함을 알 수 있습니다. 마지막으로, IF-MEM 에서 miss prediction이 발생하였을 때, epoch값을 바꾸면서 pc를 nextPC의 값으로 업데이트 함과 동시에, BTB 업데이트 동작 또한 함께 이뤄져야 할 것 입니다.

5.3 Implementing BTB Branch Prediction Module

lab6/lib/common/Btb.bsv에는 구현해야할 BTB branch prediction module의 뼈대 코드가 정의되어 있습니다.

Figure 10에서와 같이 뼈대 코드에는 단순히 'pc + 4'를 통해 예측이 이뤄집니다. 모듈 상단에 선언된 레지스터 파일들(tagArr, targetArr)은 각각 업데이트 메소드의 입력으로 들어오는 pc와, 그 다음에 fetch 해야 하는 next pc를 저장합니다. LineIdx는 파일의 가장 상단에 정의된 바와 같이 Bit#(4) type이며, 따라서 각 레지스터 파일은 16개의 entry를 저장할 수 있습니다. 레지스터 파일을 읽는 방법과 레지스터 파일의 한 인덱스에 쓰는 방법은 다음과 같습니다 (더 자세한 정보는 bluespec reference guide를 참조).

```
if(nextPC.notEmpty) begin
    pc <= nextPC.first;
    fEpoch <= !fEpoch;
    nextPC.deq;
end
else begin
    pc <= pc + 4;
end
```

Figure 8: IF-DEC stage에서 ‘‘pc + 4’’로 branch prediction 할 경우의 pc 업데이트

```
if(nextPC.notEmpty) begin
    pc <= (pc value of nextPC.first );
    fEpoch <= !fEpoch;
    bpred.update(new_pc, new_ppc)
    // new_pc: pc value of nextPC.first
    // new_ppc: ppc value of nextPC.first);
    nextPC.deq;
end
else begin
    pc <= ppc;
end
```

Figure 9: IF-DEC stage에서 BTB를 이용하여 branch prediction 할 경우의 pc 업데이트

```
typedef 4 LineNum;
typedef Bit#(LineNum) LineIdx;

...

module mkBTB(NextAddressPredictor);
    RegFile#(LineIdx, Addr) tagArr     <- mkRegFileFull;
    RegFile#(LineIdx, Addr) targetArr  <- mkRegFileFull;

    method Addr prediction(Addr pc);
        return pc + 4;
    endmethod
    mehod Action update(Addr pc, Addr target);
        noAction;
    endmoethod
endmodule
```

Figure 10: BTB branch prediction module의 뼈대 코드

앞에서도 언급하였 듯, branch prediction module에서 prediction 메소드가 불리면 BTB(여기서는 레지스터 파일들)을 참조하여 입력값의 pc에 해당하는 entry가 존재하는 지를 확인합니다. 만약 pc와 같은 entry가 존재한다면 함께 저장된 next pc 값을 리턴하고, 존재하지 않는 다면 ‘pc + 4’를 리턴하도록 합니다. 또한 update 메소드가 불리면 입력값으로 들어온 pc와 ppc를 같은 인덱스에 저장하도록 합니다. Entry의 빠른 검색을 위해, 저장할 entry의 인덱스를 pc의 일부 비트로 결정하고(truncate와 같은 함수 이용), 그 인덱스에 pc와 ppc를 저장하도록 합니다. 이렇게 하면 어떤 pc가 BTB에 존재하는 지를 찾기 위해서 한 번의 레지스터 파일(pc를 저장해놓은 레지스터 파일)을 읽는 동작과, 그 레지스터 파일에 저장되어 있는 pc가 인자로 들어온 pc와 같은지를 알아보는

```
RegFile#(Idx, Data)   regFile   <- mkRegFileFull;

Idx index = 4;
Data readData = regFile.sub(index);
//read the data of the fourth entry

Data writeData = ...;
regFile.update(5, writeData);
//write the "writeData" on the fifth entry
```

Figure 11: 레지스터 파일의 사용예

비교하는 동작 한 번만이 필요하게 됩니다.

위와 같은 과정을 거쳐 branch prediction module까지 구현하게 되면 초기에 bypassing이나 branch predictiond을 `'pc + 4'`로 할 때 보다 그 성능이 크게 향상됩니다. 구현 결과는 마찬가지로 `./benchmark_script`를 통해 확인 할 수 있으며, 완성된 프로세서를 이용하여 얻은 IPC값은 아래와 같습니다.

Bench mark	Number of instuctions	Clock cycles	IPC
median	5671	6298	0.900
multiply	21194	21957	0.965
qsort	21622	22825	0.947
towers	9903	10199	0.971
vvadd	3012	3018	0.998

Table 3: Branch prediction module 까지 적용한 프로세서의 IPC

Exercise [3] (30 points): `lab6/lib/common/Btb.bsv`에 정의된 BTB branch prediction 모듈을 완성하시오.