# Pipeline

Homework 3
모바일시스템공학과
32143153 이대훈

1. Project Introduction
Single cycle을 사용하여 Multi cycle인 파이프라인을 만든다.
파이프라인은 single cycle과 달리 하나의 instruction이 각각의 stage에서 동시에 실행한다.
Assembly code에서 사용하는 register와 instruction을 사용하여 execute한다.
결과 값은 Register number 2에 저장한다.

2. Motivation
Single cycle에서 했던 방식을 변환하여 파이프라인이 되도록 동시에 instruction을 돌리기 위해 추가적인 작업들을 해주어야 했다. 이를 통해 cycle은 늘어나지만 각각이 동작하는 clock은 줄일 수 있는 효율적인 방법을 알 수 있다.

3. Concepts used in Multi cycle
Single cycle을 바탕으로 한다.
(Mips green shit의 실행문 사용
Instruction을 각각 bit masking을 사용한 의미 확인
htonl을 사용한 리틀 엔디안에서 빅 엔디안으로 변경된 값 저장
Data path를 통한 각각의 함수 사용(Fetch, Decode, Execute, Memory, WriteBack)
Control 신호에 따른 각각의 역할 및 mux 사용)
추가)
Fetch에서 pc값을 변환해준다.(Instruction이 매 stage에서 연속적으로 들어가므로 Fetch에서 pc값을 4씩 증가시켜주어야 한다.)
Structure를 사용하여 각각의 단계에서 다음 단계로 진행할 때 Latch를 사용하여 값을 넘겨주었다.(각각 latch[0]은 전 단계의 out이고 latch[1]은 다음 단계에 들어가는 input값이다.)
모든 stage가 다 끝난 후 각각의 latch들을 update 해준다.
J type은 Decode 단계에서 실행해준다.(J-type 뒤에는 nop이 따라오기 때문에 J type이 Decode에 있을 때 Fetch에는 nop이 들어가 있어서 값에 변화를 주진 않는다.)
Data Dependancy는 Forwarding을 사용하여 해결해주었다.
Branch는 Flush를 사용하여 해결한다.

4. Program Structure
main fuction에서 두 개의 while loop을 사용
전역 변수 선언

Structure 선언 및 member 입력
Open file
---
각각의 instruction을 memory에 저장
---
Fetch
Decode(control, J type)
Exe(Data dependancy)
Memory
WriteBack

Latch Update
---
Fetch : Memory에 저장한 instruction을 하나씩 가지고 온다. pc값을 4씩 증가 시켜준다.
Decode : Fetch를 통해 가져온 instruction을 opcode를 보고 각각의 맞는 type에 따라 bit masking을 통해 구분해준다. (type : R, I, J)
Decode를 통해 생성한 값들을 통해 control 값들을 설정한다.
J type을 실행한다.
Exe : Data dependancy를 통해 이 전에 나온 결과 값을 Writeback까지 거치지 않고 미리 사용한다.(Forwarding)
Decode를 통해 각 type마다 가지고 있는 값을 사용하여 프로그램의 연산을 실행해준다.
Memory : Control에 따른 조건으로 sw와 lw 등에서 memory를 사용할 때 각각 결과를 출력해준다.
WriteBack : 모든 연산을 거친 후 결과 값은 다시 register에 저장하여야 한다.

## 5. Problems and solutions

파이프라인에서 각각의 단계에서의 값들을 사용하기 위해서는 전역변수를 사용하면 안 된다. 각각의 stage가 동시에 진행되기 때문에 그때 사용하기 위해 structure를 사용하여 그때에 맞는 값이 사용되도록 하였다.
main에서 while문을 사용할 때 조건에 따라 cycle수가 달라진다. Single cycle에서는 각각의 instruction이 실행하면 마지막 instruction이 끝날 때 cycle은 끝난다. 하지만 파이프라인에서는 마지막 instruction이 Fetch에 들어가서 WriteBack까지 끝나야지만 완벽히 끝난다. 따라서 이 문제를 해결해주기 위해 Fin이라는 전역변수를 만들어 주었다.
Data dependancy를 사용할 때 Forwarding을 통해 사용하였다. 이 때 처음에는 조건문을 if와 else if를 통해 나누었다. 하지만 이러한 방법은 error가 발생하였다. 만약 if문이 성립한다면 else if는 실행되지 않는다. 여기서 처음 if문에는 distance 2에 대한 조건이었고 else if는 distance 1에 대한 조건이었다. 값이 update 될 때에는 더 가까운 최신 값을 사용해야 하므로 distance1과 2가 모두 성립한다면 1에서 최신 값을 가져오는 것이 올바른 방법이다. 그래서 이러한 문제를 해결하고자 distance 1이 뒤에 오도록 하여 if문으로 모두 바꿔주었다. 이러한 경우 distance2가 실행한 후 distance1을 실행하므로 최신 값이 덮어 쓰여 지는 현상이 일어나 가장 근래의 값으로 계산할 수 있다. 좀 더 나아가 distance 1을 if문에 두고

distance2를 else if 문에 사용하여 해결하였다. 또한 distance3이상을 위해 값이 Register에 먼저 저장하도록 Writeback을 Fetch 다음 순서로 두었다. 이를 통해 Writeback을 먼저 해 값을 먼저 Register에 저장하고 그 이후에 저장된 Register값을 사용할 수 있다.
Branch에서는 stall대신 flush를 사용하였다. Branch를 하고나면 다음 Fetch에서 들어오는 instruction의 pc값이 4가 증가된 Branch 다음 instruction이거나 Branch 조건이 성립되어 뛰는 pc + 4 + Branchaddr인 instruction이 들어와야 한다. 이 때 Branch에 대한 조건을 보는 것은 Execution단계이다. 파이프라인에서는 계속적으로 값이 들어오기 때문에 Branch가 Execution단계에 올 때까지 두 개의 instruction이 각각 Fetch와 Decode에 들어온다. 여기서 Branch 조건이 성립하지 않는다면 그대로 이미 들어와 있던 instruction을 사용하면 되지만 Branch가 성립한다면 이미 들어온 두 개의 instruction에 대해 어떠한 영향도 받으면 안 된다. 이를 해결하기 위해 memset이라는 함수를 통해 각각의 instruction을 nop과 같은 역할을 하도록 모든 값이 0이 되도록 만들어 주었다. 이러면 이전의 두 개의 instruction은 이전의 값들에 아무 영향을 주지 않는다.
코드를 디버깅해서 출력하는 과정에서 Register에 들어있는 값이 mips.asm 파일을 해석한 것과 다른 경우가 많았다. 이런 경우 먼저 pc값들을 확인해서 pc가 알맞게 진행하는지를 확인하였고 특히 주소 값이 들어간 경우가 많았는데 문제를 해결하기 위해 값이 틀린 cycle에서 각각의 단계마다 printf를 사용하여 값을 출력하였고 이를 통해 어떤 stage에서 값이 틀려지는지를 알아냈다. 이 과정에서 LW에 대한 계산과정이 잘못되었다는 것을 발견하고 고칠 수 있었다.

## 6. Build enviroment

Compilation : Linux Assam, with GCC
To compile : cd test_prog
gcc multi_basic.c , multi_ABT.c , multi_ANT.c
To run :
./a.out
input file : simple.bin, simple2.bin, simple3.bin, simple4.bin, fib.bin, gcd.bin, fib2.bin(jalr) input4.bin

## 7. Personal feelings

이번 과제를 하면서 파이프라인에 대해 많이 배우고 또한 single cycle에서 미흡하게 사용했던 control signal에 대해서 좀 더 명확히 사용하는 시간이었다. 코드를 작성하는 과정에서는 single cycle에서 했던 것과 많이 비슷한 부분이 많아 오히려 시간은 줄일 수 있었다. 하지만 처음 나오는 기능인 Data dependancy와 J type의 실행 위치 변환, Branch에 대해서 많은 시간이 걸렸다. 하지만 덕분에 code에서 불필요한 cycle을 줄이기 위해 더 많이 노력하며 이해할 수 있었다. 이 외에도 structure 사용, while문 종료 조건 등의 작업을 거쳤다. 하지만 가장 어려운 부분은 code를 디버깅 하는 과정이었다. 5개의 stage가 모두 동시에 돌아가다 보니 오류인 곳도 찾기 어려울 뿐더러 Data dependancy까지 생각해주어야 했다. 이를 통해 mips.asm 파일들을 항상 비교하며 해석하였다. 덕분에 작년 2학년 2학기 때 들었던 시스템 프로그래밍 수업을 많이 떠올리게 되었다.

## 8. Branch prediction

Branch prediction을 사용하기 위해서 Targetpc와 Targetaddr를 설정하였다. 처음 Branch instruction이 Fetch가 되면 이때에는 Branch prediction을 하지 못한다. 이 Branch instruction이 Decode에서 opcode를 확인하고 이 때 Branch instruction인지를 확인해서 이 경우에 대해 pc값과 뛰는 위치인 pc + 4 + Branchaddr를 각각 Targetpc와 Targetaddr에 저장한다. 이렇게 하면 다음 Branch instruction이 Fetch에 들어오면 그 때의 pc값과 Targetpc값이 같아지므로 Branch prediction을 실행할 수 있다. 이때 all ways taken이면 pc는 Targetaddr로 되고 all ways not taken 이면 pc는 pc + 4가 된다. 이를 통해 Branch prediction을 사용하였다.

9. Screen capture

main함수 내의 while문

```
148         while(Fin != 0xFFFFFFFF)
149         {
150                 cycle++;
151                 printf("--------------------cycle : %d--------------------\n", cycle);
152                 printf("PC = 0x%X\n", pc);
153                 Fetch();
154                 printf("Fetch : pc = 0x%X, inst = 0x%X\n", ifid_latch[0].pc, ifid_latch[0].inst);
155                 WriteBack();
156                 printf("Decode : pc = 0x%X, inst = 0x%X\t", idex_latch[0].pc, idex_latch[0].inst);
157                 Decode();
158                 Exe();
159                 Memory();
160
161                 ifid_latch[1] = ifid_latch[0];
162                 idex_latch[1] = idex_latch[0];
163                 exmm_latch[1] = exmm_latch[0];
164                 mmwb_latch[1] = mmwb_latch[0];
165
166                 Total = Rtype + Itype + Jtype;
167         }
```

while문 조건에서 전역변수 pc가 -1이될 때 종료를 하면 마지막 instruction이 Execution일 때 파이프라인이 끝난다. 이러면 2개의 cycle을 더 보일수 없어서 Fin이라는 변수를 통해 설정하였다.

Data dependancy

```
263
264         if((idex_latch[1].rs != 0) && (idex_latch[1].rs == exmm_latch[0].v3) && (exmm_latch[0].Reg
    Write))
265         {
266                 idex_latch[1].v1 = exmm_latch[0].res;
267         }
268         else if((idex_latch[1].rs != 0) && (idex_latch[1].rs == mmwb_latch[0].v3) && (mmwb_latch[0
    ].RegWrite))
269         {
270                 if(mmwb_latch[0].MemtoReg == 1)
271                 {
272                         idex_latch[1].v1 = mmwb_latch[0].Mres;
273                 }
274                 else
275                 {
276                         idex_latch[1].v1 = mmwb_latch[0].res;
277                 }
278         }
279
280         if((idex_latch[1].rt != 0) && (idex_latch[1].rt == exmm_latch[0].v3) && (exmm_latch[0].Reg
    Write))
281         {
282                 idex_latch[1].v2 = exmm_latch[0].res;
283         }
284         else if((idex_latch[1].rt != 0) && (idex_latch[1].rt == mmwb_latch[0].v3) && (mmwb_latch[0
    ].RegWrite))
285         {
286                 if(mmwb_latch[0].MemtoReg == 1)
287                 {
288                         idex_latch[1].v2 = mmwb_latch[0].Mres;
289                 }
290                 else
291                 {
292                         idex_latch[1].v2 = mmwb_latch[0].res;
293                 }
294         }
295
```

Data dependancy에서 처음에 Distance1에 대한 조건을 비교해 Distance2보다 더 최신 값을 불러오게 하였다.

Jalr

```
359                 case 0x7: //jalr
360                         Reg[31] = idex_latch[1].pc + 4;
361                         pc = idex_latch[1].v1;
362                         memset(&ifid_latch[0], 0, sizeof(ifid));
363                         printf("Reg[31] = 0x%X, pc = 0x%X\n", Reg[31], exmm_latch[0].res);
364                         break;
```

single cycle때 했던 jalr을 사용하였다.

Branch

```
381         case 0x4: //beq
382                 if(idex_latch[1].v1 == idex_latch[1].v2)
383                 {
384                         memset(&ifid_latch[0], 0, sizeof(ifid));
385                         memset(&idex_latch[0], 0, sizeof(idex));
386                         pc = idex_latch[1].pc + 4 + idex_latch[1].BranchAddr;
387                         Rtype = Rtype - 2;
388                         TB++;
389                 }
390                 else
391                 {
392                         NB++;
393                 }
394                 printf("beq : R[%d] == R[%d]\n", idex_latch[1].rs, idex_latch[1].rt);
395                 break;
396         case 0x5: //bne
397                 if(idex_latch[1].v1 != idex_latch[1].v2)
398                 {
399                         memset(&ifid_latch[0], 0, sizeof(ifid));
400                         memset(&idex_latch[0], 0, sizeof(idex));
401                         pc = idex_latch[1].pc + 4 + idex_latch[1].BranchAddr;
402                         Rtype = Rtype - 2;
403                         TB++;
404                 }
405                 else
406                 {
407                         NB++;
408                 }
409                 printf("bne : R[%d] != R[%d]\n", idex_latch[1].rs, idex_latch[1].rt);
410                 break;
```

Branch에서 조건문이 맞으면 pc 값이 pc + 4 + branchaddr로 된다. 이때 branch가 execution까지 올 때 이전 stage인 Fetch와 Decode에는 필요 없는 instruction이 사용되었으므로 그 두 개를 flush시켜 아무런 역할을 하지 않도록 한다.

추가)

Branch prediction

1. All ways Branch taken

```
28 //Branch prediction
29 int Targetpc1 = -1;
30 int Targetadd1;
31 int Targetpc2 = -1;
32 int Targetadd2;
33 int valid1;
34 int valid2;
35
```

변수를 설정해준다. 이때 Targetpc는 처음 0이 들어가면 처음 전역변수 pc와 같아

아래와 같은 경우의 조건문에서 무한히 사용된다. 따라서 pc 값이 4씩 증가하지 않고 0으로 유지된다. 이런 문제를 해결해주기 위해 처음 Targetpc를 -1로 초기화하고 시작한다.

```
200         else if(pc == Targetpc1)
201         {
202                 pc = Targetadd1;
203                 NB++;
204         }
205         else if(pc == Targetpc2)
206         {
207                 pc = Targetadd2;
208                 NB++;
209         }
```

Decode에서 branch instruction이 들어왔을 때 각각 branch instruction의 pc값을 Targetpc에 pc + 4 + branchaddr를 Targetaddr  저장한다. 이렇게 저장된 값이 위의 Fetch 단계에서 성립하면 pc값에 Targetaddr가 들어간다.

```
275         if((idex_latch[0].opcode == 0x4) && (idex_latch[0].pc != Targetpc1))
276         {
277                 Targetpc1 = ifid_latch[1].pc;
278                 Targetadd1 = ifid_latch[1].pc + 4 + idex_latch[0].BranchAddr;
279                 valid1 = 1;
280         }
281         if((idex_latch[0].opcode == 0x5) && (idex_latch[0].pc != Targetpc2))
282         {
283                 Targetpc2 = ifid_latch[1].pc;
284                 Targetadd2 = ifid_latch[1].pc + 4 + idex_latch[0].BranchAddr;
285                 valid2 = 1;
286         }
```

Branch instruction이 들어왔을 때 Targetpc가 정해지고 이때 다른 pc에서 Branch instruction이 다시 들어온다면 그 때에 대해 Targetpc와 Targetaddr를 다시 설정해준다.

```
411         case 0x4: //beq
412                 if(valid1 == 0)
413                 {
414                         if(idex_latch[0].v1 == idex_latch[1].v2)
415                         {
416                                 memset(&ifid_latch[0], 0, sizeof(ifid));
417                                 memset(&idex_latch[0], 0, sizeof(idex));
418                                 pc = idex_latch[1].pc + 4 + idex_latch[0].BranchAddr;
419                                 Rtype = Rtype - 2;
420                                 TB++;
421                                 valid1 = 0;
422                         }
423                 }
424                 if(idex_latch[1].v1 != idex_latch[0].v2)
425                 {
426                         memset(&ifid_latch[0], 0, sizeof(ifid));
427                         memset(&idex_latch[0], 0, sizeof(idex));
428                         pc = idex_latch[1].pc + 4;
429                         Rtype = Rtype - 2;
430                         TB--;
431                         NB++;
432                 }
433                 printf("beq : R[%d] == R[%d]\n", idex_latch[1].rs, idex_latch[1].rt);
434                 break;
435         case 0x5: //bne
436                 if(valid2 == 0)
437                 {
438                         if(idex_latch[0].v1 != idex_latch[1].v2)
439                         {
440                                 memset(&ifid_latch[0], 0, sizeof(ifid));
441                                 memset(&idex_latch[0], 0, sizeof(idex));
442                                 pc = idex_latch[1].pc + 4 + idex_latch[1].BranchAddr;
443                                 Rtype = Rtype - 2;
444                                 TB++;
445                                 valid2 = 0;
446                         }
447                 }
448                 if(idex_latch[1].v1 == idex_latch[0].v2)
449                 {
450                         memset(&ifid_latch[0], 0, sizeof(ifid));
451                         memset(&idex_latch[0], 0, sizeof(idex));
452                         pc = idex_latch[1].pc + 4;
453                         Rtype = Rtype - 2;
454                         TB--;
455                         NB++;
456                 }
457                 printf("bne : R[%d] != R[%d]\n", idex_latch[1].rs, idex_latch[1].rt);
458                 break;
```

처음 Branch instuction이 들어왔을 때는 아직 Targetpc와 Targetaddr가 설정되지 않았을 때이다. 그렇기 때문에 처음에 한해서 valid라는 변수를 사용해 execution에

서 실행하게 하고 다음부터는 사용하지 않도록 한다. 이때 all ways branch taken 이면 branch prediction을 통해 pc가 Targetaddr로 이동한다. 따라서 execution에서 branch가 성립을 안할 때 pc가 다시 pc + 4가 되고 이전에 Fetch와 Decode에 들어와 있는 값을 flush 시켜준다.

2. All ways Branch not taken

All ways Branch taken과 사용하는 방법은 매우 유사하며 이때는 not taken이므로 Targetaddr에 pc + 4가 저장된다.

```
28 //Branch prediction
29 int Targetpc1 = -1;
30 int Targetadd1;
31 int Targetpc2 = -1;
32 int Targetadd2;
33 int valid1;
34 int valid2;
35
```

```
275         if((idex_latch[0].opcode == 0x4) && (idex_latch[0].pc != Targetpc1))
276         {
277                 Targetpc1 = ifid_latch[1].pc;
278                 Targetadd1 = ifid_latch[1].pc + 4;
279                 valid1 = 1;
280         }
281         if((idex_latch[0].opcode == 0x5) && (idex_latch[0].pc != Targetpc2))
282         {
283                 Targetpc2 = ifid_latch[1].pc;
284                 Targetadd2 = ifid_latch[1].pc + 4;
285                 valid2 = 1;
286         }
```

```
197         if(pc == 0xFFFFFFFF)
198         {
199         }
200         else if(pc == Targetpc1)
201         {
202                 pc = Targetadd1;
203                 NB++;
204         }
205         else if(pc == Targetpc2)
206         {
207                 pc = Targetadd2;
208                 NB++;
209         }
210         else
211         {
212                 pc = pc + 4;
213         }
214 }
```

10. 결과 창

< multi_basic.c > - 기본 코드

simple.bin

```
Fianl : R[2] = 0
Number of cycle : 12
R-type : 4, I-type : 4, J-type : 0
Total number of instructions : 8
Memory Access : 2
Register Access : 7
Branches : 0
Not-taken branches : 0
Number of Jumps : 0
daehoon14@assam:~/test_prog$
```

simple2.bin

```
Fianl : R[2] = 100
Number of cycle : 14
R-type : 3, I-type : 7, J-type : 0
Total number of instructions : 10
Memory Access : 4
Register Access : 8
Branches : 0
Not-taken branches : 0
Number of Jumps : 0
```

simple3.bin

```
Fianl : R[2] = 5050
Number of cycle : 1537
R-type : 410, I-type : 920, J-type : 1
Total number of instructions : 1331
Memory Access : 613
Register Access : 921
Branches : 101
Not-taken branches : 1
Number of Jumps : 1
```

simple4.bin

```
Fianl : R[2] = 55
Number of cycle : 296
R-type : 110, I-type : 153, J-type : 11
Total number of instructions : 274
Memory Access : 100
Register Access : 193
Branches : 9
Not-taken branches : 1
Number of Jumps : 11
daehoon14@assam:~/test_prog$
```

gcd.bin

```
Fianl : R[2] = 1
Number of cycle : 1294
R-type : 498, I-type : 637, J-type : 65
Total number of instructions : 1200
Memory Access : 486
Register Access : 827
Branches : 45
Not-taken branches : 28
Number of Jumps : 65
daehoon14@assam:~/test_prog$
```

fib.bin

```
Fianl : R[2] = 55
Number of cycle : 3173
R-type : 1200, I-type : 1697, J-type : 164
Total number of instructions : 3061
Memory Access : 1095
Register Access : 2131
Branches : 54
Not-taken branches : 55
Number of Jumps : 164
daehoon14@assam:~/test_prog$
```

fib2.bin(Jalr)

```
Fianl : R[2] = 55
Number of cycle : 3391
R-type : 1417, I-type : 1807, J-type : 55
Total number of instructions : 3279
Memory Access : 1096
Register Access : 2457
Branches : 54
Not-taken branches : 55
Number of Jumps : 55
daehoon14@assam:~/test_prog$
```

input4.bin

```
200155945 ------------------------------------------------------------------------
200155946 Fianl : R[2] = 85
200155947 Number of cycle : 27430473
200155948 R-type : 10152965, I-type : 13219741, J-type : 103
200155949 Total number of instructions : 23372809
200155950 Memory Access : 7116606
200155951 Register Access : 18288047
200155952 Branches : 2028830
200155953 Not-taken branches : 869
200155954 Number of Jumps : 103
                                                          200155954,1   Bot
```

< multi_ABT.c > - Branch prediction all ways branch taken

simple.bin

```
Fianl : R[2] = 0
Number of cycle : 12
R-type : 4, I-type : 4, J-type : 0
Total number of instructions : 8
Memory Access : 2
Register Access : 7
Branches : 0
Not-taken branches : 0
Number of Jumps : 0
daehoon14@assam:~/test_prog$
```

simple2.bin

```
Fianl : R[2] = 100
Number of cycle : 14
R-type : 3, I-type : 7, J-type : 0
Total number of instructions : 10
Memory Access : 4
Register Access : 8
Branches : 0
Not-taken branches : 0
Number of Jumps : 0
daehoon14@assam:~/test_prog$
```

simple3.bin

```
--------------------------------------------------------------------
Fianl : R[2] = 5050
Number of cycle : 1339
R-type : 410, I-type : 920, J-type : 1
Total number of instructions : 1331
Memory Access : 613
Register Access : 1020
Branches : 102
Not-taken branches : 1
Number of Jumps : 1
daehoon14@assam:~/test_prog$
```

simple4.bin

```
--------------------------------------------------------------------
Fianl : R[2] = 55
Number of cycle : 282
R-type : 110, I-type : 153, J-type : 11
Total number of instructions : 274
Memory Access : 100
Register Access : 200
Branches : 10
Not-taken branches : 1
Number of Jumps : 11
daehoon14@assam:~/test_prog$
```

gcd.bin

```
--------------------------------------------------------------------
Fianl : R[2] = 1
Number of cycle : 1264
R-type : 498, I-type : 637, J-type : 65
Total number of instructions : 1200
Memory Access : 486
Register Access : 842
Branches : 73
Not-taken branches : 28
Number of Jumps : 65
daehoon14@assam:~/test_prog$
```

fib.bin

```
--------------------------------------------------------------------
Fianl : R[2] = 55
Number of cycle : 3177
R-type : 1200, I-type : 1697, J-type : 164
Total number of instructions : 3061
Memory Access : 1095
Register Access : 2129
Branches : 109
Not-taken branches : 55
Number of Jumps : 164
daehoon14@assam:~/test_prog$
```

fib2.bin(Jalr)

```
--------------------------------------------------------------------
Fianl : R[2] = 55
Number of cycle : 3395
R-type : 1417, I-type : 1807, J-type : 55
Total number of instructions : 3279
Memory Access : 1096
Register Access : 2455
Branches : 109
Not-taken branches : 55
Number of Jumps : 55
daehoon14@assam:~/test_prog$
```

input4.bin

```
171767361 ------------------------------------------------------------------------
171767362 Fianl : R[2] = 85
171767363 Number of cycle : 23374961
171767364 R-type : 10152965, I-type : 13219741, J-type : 103
171767365 Total number of instructions : 23372809
171767366 Memory Access : 7116606
171767367 Register Access : 20315803
171767368 Branches : 2029698
171767369 Not-taken branches : 869
171767370 Number of Jumps : 103
                                                                    171767370,1   Bot
```

< multi_ANT.c > - Branch prediction all ways branch not taken

simple.bin

```
------------------------------------------------------------------------
Fianl : R[2] = 0
Number of cycle : 12
R-type : 4, I-type : 4, J-type : 0
Total number of instructions : 8
Memory Access : 2
Register Access : 7
Branches : 0
Not-taken branches : 0
Number of Jumps : 0
daehoon14@assam:~/test_prog$
```

simple2.bin

```
------------------------------------------------------------------------
Fianl : R[2] = 100
Number of cycle : 14
R-type : 3, I-type : 7, J-type : 0
Total number of instructions : 10
Memory Access : 4
Register Access : 8
Branches : 0
Not-taken branches : 0
Number of Jumps : 0
daehoon14@assam:~/test_prog$
```

simple3.bin

```
------------------------------------------------------------------------
Fianl : R[2] = 5050
Number of cycle : 1537
R-type : 208, I-type : 920, J-type : 1
Total number of instructions : 1129
Memory Access : 613
Register Access : 719
Branches : 202
Not-taken branches : 101
Number of Jumps : 1
daehoon14@assam:~/test_prog$
```

simple4.bin

```
------------------------------------------------------------------------
Fianl : R[2] = 55
Number of cycle : 296
R-type : 92, I-type : 153, J-type : 11
Total number of instructions : 256
Memory Access : 100
Register Access : 175
Branches : 18
Not-taken branches : 9
Number of Jumps : 11
daehoon14@assam:~/test_prog$
```

gcd.bin

```
Fianl : R[2] = 1
Number of cycle : 1294
R-type : 408, I-type : 637, J-type : 65
Total number of instructions : 1110
Memory Access : 486
Register Access : 737
Branches : 90
Not-taken branches : 71
Number of Jumps : 65
daehoon14@assam:~/test_prog$
```

fib.bin

```
Fianl : R[2] = 55
Number of cycle : 3173
R-type : 1092, I-type : 1697, J-type : 164
Total number of instructions : 2953
Memory Access : 1095
Register Access : 2023
Branches : 108
Not-taken branches : 108
Number of Jumps : 164
daehoon14@assam:~/test_prog$
```

fib2.bin(Jalr)

```
Fianl : R[2] = 1
Number of cycle : 3282
R-type : 1201, I-type : 1806, J-type : 55
Total number of instructions : 3062
Memory Access : 1095
Register Access : 2241
Branches : 108
Not-taken branches : 108
Number of Jumps : 55
daehoon14@assam:~/test_prog$
```

input4.bin

```
200155945 ----------------------------------------------------------------------
200155946 Fianl : R[2] = 85
200155947 Number of cycle : 27430473
200155948 R-type : 6095305, I-type : 13219741, J-type : 103
200155949 Total number of instructions : 19315149
200155950 Memory Access : 7116606
200155951 Register Access : 14230387
200155952 Branches : 4057660
200155953 Not-taken branches : 2029493
200155954 Number of Jumps : 103
                                                          200155954,1   Bot
```

출력

v0값을 Reg[2]에서 출력한다.

전체 cycle 수를 출력한다

R type인지 I type인지 J type인지를 출력한다. 이 때 구분을 writeback에서 하므로 처음 시작할 때 nop이 처음 instruction이 writeback에 오기까지 4번 작용한다. 따라서 R type은 초기값을 -4로 시작하였다. 또 한 branch에서 flush를 통해 만들어지는 nop은 instruction에 포함하지 않기 위해 branch가 성립할 때 R type에서 2를 빼주었다.

Total은 R, J, I type의 총 합이다.

memory access는 sw와 lw가 이러난 횟수이다.

Register access는 writeback이 실행한 횟수이다. 이 때 sw나 branch, jump는 writeback

을 사용하지 않으므로 개수에 포함하지 않는다.
branch가 실행한 횟수이며 all ways branch taken에서는 항상 branch가 실행했다하여 증가시켰다.
Not taken branch는 branch가 성립하지 않을 때 증가한다.
Jump가 실행하는 횟수( J, Jal)