Review for Exam
Appendix C and Chapter 3

How to design a CPU
- data paths and control paths
- Pipelined CPU
- Improving performance
- Data forwarding
- Branch prediction – static

Data hazards – RAW, WAR and WAW
- Scoreboard
- Tomasulo’s algorithm
- Dynamic Branch Prediction
- loop unrolling and reordering
- register renaming

Difference between software reordering (and loop unrolling) and hardware techniques for out of order execution (score board and reservation stations)

Consider the following example

```
LOOP:
LD   F6, 34(R2)
LD   F2, 45(R3)
MULD F0, F2, F6
SUBD F8, F0, F2
ADDD F2, F4, F6
SD   F8, 34(R2)
SD   F2, 45(R3)
ADDI R2, R2, #8
ADDI R3, R3, #8
BNE  R3, R1, LOOP
```

Let us assume the following latencies
- 2 for LD
- 5 for MULD
- 3 for ADDD/SUBD
- 1 for ADDI/BNE

Let us first do software (compiler) reordering
CSCE 4610: Computer Architecture

LD F6, 34(R2)
LD F2, 45(R3)
STALL
MULD F0, F2, F6
STALL
STALL
STALL
SUBD F8, F0, F2
ADDD F2, F4, F6
STALL
SD F8, 34(R2)
SD F2, 45(R3)
ADDI R2, R2, #8
ADDI R3, R3, #8
BNE R3, R1, LOOP

16 cycles

Without reordering

LD F6, 34(R2)
LD F2, 45(R3)
MULD F0, F2, F6
ADDI R2, R2, #8
ADDI R3, R3, #8
STALL
SUBD F8, F0, F2
ADDD F2, F4, F6
STALL
SD F8, 26(R2)
SD F2, 37(R3)
BNE R3, R1, LOOP

14 cycles

With reordering

Of course we can unroll the loop and reorder to get better performance

LD F6, 34(R2)
LD F2, 45(R3)
MULD F0, F2, F6
ADDI R2, R2, #8
ADDI R3, R3, #8
STALL
MULD F0, F2, F6
ADDI R2, R2, #16
ADDI R3, R3, #16
STALL
MULD F14, F12, F10
ADDI R2, R2, #16
ADDI R3, R3, #16
STALL
ADDI F8, F0, F2
ADDD F2, F4, F6
SUBD F16, F14, F10
SD F8, 18(R2)
SD F2, 26(R3)
SD F16, 26(R2)
SD F18, 37(R2)
BNE R3, R1, LOOP

18 cycles for 2 iterations

Do we need to worry about WAR, WAW while reordering?

For example, we cannot reorder ADDD and SUBD because of a WAR on F2
We can use register renaming if necessary

CSCE 4610 March 31, 2017

CSCE 4610: Computer Architecture

Now let us think of using Scoreboard to see how we can process the same code
Remember for scoreboard, we will keep track of cycles when an instruction is issued, when inputs are read, when execution starts and when results are written back to registers
Let us assume that we have 2 floating point adders, 1 float multiply and 1 LD/SD and a integer adder.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Issue</th>
<th>inputs read</th>
<th>execute</th>
<th>wb</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F6, 34(R2)</td>
<td>1</td>
<td>2</td>
<td>3–4</td>
<td>5</td>
</tr>
<tr>
<td>LD F2, 45(R3)</td>
<td>6</td>
<td>7</td>
<td>8–9</td>
<td>10</td>
</tr>
<tr>
<td>MULD F0, F2, F6</td>
<td>7</td>
<td>11</td>
<td>12–16</td>
<td>17</td>
</tr>
<tr>
<td>SUBD F8, F0, F2</td>
<td>8</td>
<td>18</td>
<td>19–21</td>
<td>22</td>
</tr>
<tr>
<td>ADDD F2, F4, F6</td>
<td>11</td>
<td>12</td>
<td>13–15</td>
<td>19</td>
</tr>
<tr>
<td>SD F8, 34(R2)</td>
<td>12</td>
<td>23</td>
<td>24–25</td>
<td></td>
</tr>
<tr>
<td>SD F2, 45(R3)</td>
<td>26</td>
<td>27</td>
<td>28–29</td>
<td></td>
</tr>
<tr>
<td>ADDI R2, R2, #8</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
</tr>
<tr>
<td>ADDI R3, R3, #8</td>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
</tr>
<tr>
<td>BNE R2, R1, LOOP</td>
<td>35</td>
<td>36</td>
<td>37</td>
<td></td>
</tr>
</tbody>
</table>

In order issue delays some instructions

Note that the second LD is not issued until first LD completes – structural hazard
this does not exist if we assume LD/SD is pipelined (same with ADDI)
ADDD issue is delayed due to WAW on F2 (with second LD)
ADDD write back is delayed due to WAR on F2 (with SUBD)
We can eliminate some dependencies with register renaming

CSCE4610 March 31, 2017
If we are using Tomasulo’s approach, we do not have to worry about WAR and WAW. Also using reservation stations, we can issue instructions to the same functional unit, and we can assume the functional units to be pipelined.

<table>
<thead>
<tr>
<th>Issue</th>
<th>Inputs</th>
<th>Execute</th>
<th>CDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F6, 34(R2)</td>
<td>1</td>
<td>2</td>
<td>3-4</td>
</tr>
<tr>
<td>LD F2, 45(R3)</td>
<td>2</td>
<td>3</td>
<td>4-5</td>
</tr>
<tr>
<td>MULT F6, F2, F6</td>
<td>3</td>
<td>6</td>
<td>7-11</td>
</tr>
<tr>
<td>SUBD F8, F0, F2</td>
<td>4</td>
<td>12</td>
<td>13-15</td>
</tr>
<tr>
<td>ADDD F2, F4, F6</td>
<td>5</td>
<td>6</td>
<td>7-9</td>
</tr>
<tr>
<td>SD F8, 34(R2)</td>
<td>6</td>
<td>16</td>
<td>17-18</td>
</tr>
<tr>
<td>SD F2, 45(R3)</td>
<td>7</td>
<td>10</td>
<td>11-12</td>
</tr>
<tr>
<td>ADDI R2, R2, #8</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>ADDI R3, R3, #8</td>
<td>9</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>BNE R2, R1, LOOP</td>
<td>10</td>
<td>13</td>
<td>14</td>
</tr>
</tbody>
</table>

Separate LD/SD
And integer unit
CDB can be broadcast as soon as result is available.

I am assuming that you can read at the same time the results are placed on CDB.

It is important to understand the hardware and software solutions and how and who takes care of WAW and WAR dependencies.

We also talked about register renaming. In our software solution using loop unrolling, we manually renamed registers.

If we allowed hardware to do register renaming, scoreboard technique may have produced better results – since we could have eliminated WAR and WAW dependencies and issued instructions sooner, and written results back sooner.

Of course you can use both hardware and software techniques to further improve performance.
Example problems

Design a different type of a pipeline
   We have seen how to implement JAL
   Consider a different way of implementing function call
   JSR – store the return address on stack
       Stack pointer is Register R-29

Actions to perform
   During Decode read R29
   During execute, add PC+4 to (shift left by 2)(Sign extend) IR(0-15)
       compute address of the function to jump to
   During Memory store PC+4 at MEM[R29]
       store return address on stack
   Increment R29 by 4 (can be done either in execute using a separate adder)
   In write-back,
       Store R29 (updated)
       Modify PC to procedure address
Another example of designing pipelines for the following type of instructions

\[ \text{Add } a(R_d), b(R_{s1}), c(R_{s2}) \]

We need 64 bit instructions (we can use 14-bits for each offset, 5 bits for each register field and 7 bit opcode)

We saw the solution to this problem in class

---

IF: Fetch the instruction fetching. No change
Decode: Use the instruction and read \( R_{s1} \) and \( R_{s2} \)

ALU-1: \( b+R_{s1} \). We need an integer adder
ALU-2+Mem: \( b+R_{s2} \) At the same time read memory to get first source
ALU-2+MEM-1: \( b+R_{s2} \) At the same time read memory to get first source
ALU-2-MEM-2: Read second operand from memory, and compute \( c+R_d \)
ALU-3: Add the two data items read from memory
MEM-3: Store result from ALU-3 into memory

How many register ports and memory ports are needed?

Since Registers are read only during ID, we need only 3 read ports

We are ignoring for now on how we modify register values. We need additional register ports if we have instructions that modify registers

We access memory in MEM-1, MEM-2, MEM-3, and we need 2 read and one write memory ports.
Another example of redesigning pipelines

How do we implement indirect memory addressing?
LWI Rd, disp

This instruction uses disp as an indirect address. That is, use disp as a memory address, read the contents of memory at that address, and use the value just read as the address of the real operand (that is, read memory again and store the value in Rd).

For example: LWI R1, 100 means, first read memory location 100, and use the value read (say 401) as an address to memory again, read that memory location (401). The value read from 401 will be stored in Rd.

How do we change our pipelines to allow this mode?
Another problem about implementing different instruction type

Many signal processing applications involve computations of the type $\sum a[i] \cdot b[i]$

Some architectures have instructions known as Multiply-Accumulate to help such computations. Consider for example the following instruction

```
MAC R_i, R_j, R_k : R_i = R_i + R_j \cdot R_k
```

Show how you can modify the 5-stage pipeline to implement MAC instruction. You can add additional stages if you feel 5 stages are not sufficient. It is better to first describe all the actions needed to implement MAC and then show details of data paths.
**Key.** Let us first discuss the actions to be performed.

1. Remember this instruction actually has 3 input arguments: \( R_i \), \( R_j \) and \( R_k \). So we need to read all three registers in Decode stage *(we need 3 read ports to registers)*

2. We need to perform two arithmetic operations.
   
   So we can think of using two execute stages.
   
   In the first EX-1 stage, we will perform the multiplication, and in the second EX-2 stage we will perform the addition.
   
   For instructions that perform only one arithmetic operation
   
   (say \( \text{ADD } R_i, R_j, R_k \) where we are adding \( R_j \) to \( R_k \) and storing the result in \( R_i \)),
   
   we pass through EX-2 without performing any operations.

3. We can perform MEM operation in EX-2. We can then call this stage EX-2-MEM.

4. In write-back, WB stage, we store \( R_i + R_j \times R_k \) back in \( R_i \) *(we need one write port)*
Consider the following code segment in MIPS:

```
LOOP:    LD    F2, 100 (R2)    : Load to F2 value from memory
        ADDD  F4, F2, F8    : F4 = F2+F8
        MULD  F6, F4, F10  : F6 = F4*F10
        SD    100 (R2), F6   : store F6 to memory
        ADDDI R2, R2, #8    : R2 = R2+8
        BNE   R2, R6, Loop
```

Assume that you have the normal 5 stage pipeline (Instruction Fetch, Instruction Decode, Execute, Memory and write-back). Also we will use the following latencies:

- Integer operations: 1 cycle
- LD and STORE: 2 cycles
- Floating Add: 3 cycles
- Floating Mult: 5 cycles

a) Show how many cycles are needed to complete one iteration of the loop without reordering the code.
b) Use delayed branch and show the number of cycles needed. You can also reorder the code if you see opportunities.

**NOTE** WE ARE NOW USING SOFTWARE REORDERING – NO HARDWARE OUT OF ORDER

---

**CSCE 4610: Computer Architecture**

a). Note that the latencies indicate when the results are available for use. So the results of LD can be used 2 cycles after LD starts and if we assume the data can be forwarded to the instruction that needs the results, we only need one stall after LD before ADDD can use the LD results.

```
LOOP:    LD    F2, 100 (R2)   
        Stall
        ADDD  F4, F2, F8    
        Stall
        Stall
        MULD  F6, F4, F10   
        Stall
        Stall
        Stall
        SD    100 (R2), F6   
        Stall
        ADDDI R2, R2, #8    
        Stall
        BNE   R2, R6, Loop
```

I included a stall after BNE since we do not know if the branch will be taken or not. With this we have 14 Cycles to complete one iteration.
b). We can move SD into the delayed branch slot and also reorder some instructions to reduce the number of cycles needed.

We have to modify the offset of SD instruction since R2 is incremented by 8.

```
LOOP:    LD       F2, 100 (R2)
         Stall
         ADDD     F4, F2, F8
         Stall
         Stall
         MULD     F6, F4, F10
         Stall
         Stall
         ADDDI    R2, R2, #8
         BNE      R2, R6, Loop
         SD       92 (R2), F6
```

Now we need 11 cycles to complete one iteration.

---

More problems

In a standard 5-stage pipeline, the decision about a branch is known in the 3rd stage (Execute). Consider two choices for branch instructions

a). Use delayed branch instruction with one delay slot (i.e., one instruction after the branch will be executed) and stop fetching any additional instructions upon discovering a branch instruction.

b). Use a delayed branch with 2 delay slots (i.e., two instructions after the branch will be executed).

In the first case, we have a stall on a branch and lose one cycle (even if we save a cycle because of the delay slot). In the second case there will be no stalls. However, it is more difficult to find two useful instructions that can be placed after a branch (and fill the 2 delay slots). If we cannot find a useful instruction for a delay slot, we use a NOP and for the purpose of this example, we will assume that it is a wasted slot.

Compare these two alternatives if 20% of all instructions are branches, and an optimizing compiler can find useful instructions for one delay slot 80% of the time and find useful instructions to fill 2 delay slots only 25% of the time.
CSCE 4610: Computer Architecture

Key
Consider the case with 1 delay slot. We still lose one cycle even if we can use the delay slot. Or we will lose 2 cycles if we cannot use the delay slot. Since we can use only 80% of the time one delay slot, we have:

\[20\% \times 1 \times 80\% + 20\% \times 2 \times 20\% = 0.16 + 0.08 = 0.24 \text{ cycles lost}\]

In the second case, if we can use both delay slots, we have no loss of cycles; if only one delay slot can be used, we have a loss of one cycle and if both delay slots cannot be used we have a loss of two cycles. Thus we have:

\[20\% \times [(100 - 25)\% \times (80\% \times 1 + 20\% \times 2)]\]
\[= 0.18 \text{ cycles lost.}\]

Use Scoreboard technique and show the contents of Instruction Status and Functional Unit Status tables for two snapshots

a). At the initial state when no instruction has completed execution

b). When LD and ADDD have completed.

Remember for Scoreboard we will assume one Integer unit (for LD, SD and Integer arithmetic instructions), 2 Floating point Multiply units, one Floating point Add unit and one Floating point Divide unit. For your convenience, I am giving you the template for Instruction Status and Functional Unit status tables.

<table>
<thead>
<tr>
<th>Instruction Status</th>
<th>Issue</th>
<th>Read</th>
<th>Operands</th>
<th>Execute</th>
<th>Write results</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F2, 100(R2)</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDD F4, F2, F8</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MULD F8, F4, F10</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SD 100(R2), F6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDD R2, R2, #8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BNE R2, R6, Loop</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Functional Unit Status</th>
<th>Busy</th>
<th>Op</th>
<th>P1</th>
<th>P2</th>
<th>K1</th>
<th>K2</th>
<th>Q1</th>
<th>Q2</th>
<th>Rj</th>
<th>Rk</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>Yes</td>
<td>Load</td>
<td>F2</td>
<td>R2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MULT-1</td>
<td>Yes</td>
<td>Mult</td>
<td>F6</td>
<td>F4</td>
<td>F10</td>
<td>ADD</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MULT-2</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD</td>
<td>Yes</td>
<td>Add</td>
<td>F4</td>
<td>F2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIV</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note that we show No under Rj for Load instruction since in this snapshot we already read R2 and we want to indicate that WAR is no longer an issue (if there is an instruction waiting to write to R2, it can proceed).
Examples related dynamic instruction scheduling

Use of scoreboard – show snapshots for the following code

DIVD F0, F2, F4
ADDD F10, F0, F6
ADDD F12, F8, F14
SUBD F16, F0, F18
MULTD F0, F10, F20

I will show in which cycle count each instruction enters different states (instead of a snapshot)
CSCE 4610: Computer Architecture

Instruction status (assuming each instruction takes one cycle)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Issue</th>
<th>read ops</th>
<th>exec</th>
<th>writeback</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIVD F0, F2, F4</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>ADDD F10, F0, F6</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>ADDD F12, F8, F14</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>SUBD F16, F0, F18</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>MULTD F0, F10, F20</td>
<td>3</td>
<td>8</td>
<td>9</td>
<td>16</td>
</tr>
</tbody>
</table>

Change this assuming different execution times

Say DIVD takes 10 cycles, ADDD/SUBD takes 3 cycles, and MULTD takes 5 cycles

---

CSCE 4610: Computer Architecture

Using Tomasulo’s approach

Instruction status

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Issue</th>
<th>exec</th>
<th>cdb</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIVD F0, F2, F4</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>ADDD F10, F0, F6</td>
<td>2</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>ADDD F12, F8, F14</td>
<td>3</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>SUBD F16, F0, F18</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>MULTD F0, F10, F20</td>
<td>4</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

Reservation stations

<table>
<thead>
<tr>
<th>Status</th>
<th>OP</th>
<th>vj</th>
<th>vk</th>
<th>qj</th>
<th>qk</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mul1</td>
<td>busy</td>
<td>DIVD</td>
<td>F2</td>
<td>F4</td>
<td></td>
</tr>
<tr>
<td>Mul2</td>
<td>busy</td>
<td>MUL</td>
<td>F10</td>
<td>F20</td>
<td>add1</td>
</tr>
<tr>
<td>Add1</td>
<td>busy</td>
<td>ADD</td>
<td>F0</td>
<td>F6</td>
<td>Mul1</td>
</tr>
<tr>
<td>Add2</td>
<td>busy</td>
<td>ADD</td>
<td>F8</td>
<td>F14</td>
<td></td>
</tr>
<tr>
<td>Add3</td>
<td>busy</td>
<td>SUB</td>
<td>F0</td>
<td>F18</td>
<td>Mul1</td>
</tr>
</tbody>
</table>

---
Determine data dependencies

1: DIVD F0, F2, F4  1 to 5 WAW
2: ADDD F6, F0, F8  1 to 2 RAW
3: SD 0(R1), F6     2 to 3 RAW
4: SUBD F8, F6, F14  2 to 4 RAW
5: MULD F0, F10, F8  4 to 5 RAW

Scheduling in delay slots
Need to make sure that the instruction in delay slot does not have any dependencies with the branch instructions

Examples using branch prediction

While fetching an instruction, we also look up this BTB.

**BTB should contain a valid entry only if the instruction fetched is a branch instruction**

Otherwise there should no valid entry

Let us look at the example (page 123). We are given

- Prediction correctness = 90%
- Hit rate (found in BTB) = 90%
- Penalty of mis-prediction = 2 cycles
- Penalty if not found in BTB and taken = 2 cycles
- If not found in the BTB, and not taken, no penalty

10% misprediction for 90% found in BTB, penalty = 0.9*0.1*2 = 0.18 cycles
10% miss in BTB and 60% taken, penalty = 0.1*0.6*2 = 0.12 cycles
Total penalty = 0.3 cycles

Without branch prediction, we lose 60%*2 = 1.2 cycles
CSCE 4610: Computer Architecture

We are using branch target buffer (with pre-computed branch addresses).
The mis-prediction penalty is 4 cycles
Buffer miss penalty is 3 cycles (either taken or not taken)
90% hit rate (of finding in the buffer) and 90% correct prediction rate
15% of all instructions are branches

What is the CPI?

Consider a miss in BTB
Stalls = 10%*3 = 0.3
Hit in BTB but mis-prediction
Stalls = 90%*10% *4 = 0.36
Total stalls = 0.66
But we only have 15% branch instructions
CPI = 1+ 15%*0.66 = 1.099

What about if we do not use BTB and assume penalty for branches is 3 cycles
(we just stall when we see a branch)
Stalls = 15%*3 = 0.45 cycles
CPI = 1.45
Speed up = 1.45/1.099 = 1.319 or 31.9%

We are given 15% of all instructions are conditional branches
1% unconditional
60% of conditional branches are taken

No penalty for not taken branches
2 cycles for taken (including unconditional) branches

Stalls = 15%*60%* 2 + 1%*2 = 0.18+ 0.02 = 0.2 cycles
Consider the following state transition diagram representing a different branch prediction technique (different from the 2-bit predictor). We still need two bits.

![State Transition Diagram]

a). Describe the behavior of this branch prediction method in detail. How is this different from standard 2-bit predictor?

b). Under what assumptions about the program behavior does this type of a branch prediction leads to better performance than a standard 2 bit predictor?

Key. Note that this prediction favors Predict taken. This taken prediction is not changed until 2 mis-predictions. However, the predict not-taken the prediction is changed to predict taken on one misprediction.

However, at this time, we change predictions on every misprediction (unless two predict taken branches are correctly predicted).

b). A program with many nested if..then .. if ..then.. else if.. then.. type structures achieve better performance with the new prediction (where you predict the condition to be true, but occasionally the else part will be executed, followed by another if).
Consider the following two state transition diagrams representing two different branch prediction techniques.

a). The branch prediction on the left (A) favors taken branches. Note that you keep “predict-taken” until this leads to two miss predictions in a row. However, the prediction, “predict-not-taken” is changed to “predict-taken” on one miss prediction.

The diagram on the right (B) is a minor variation of the standard 2-bit predictor. You change your prediction (for both predict-taken and predict-not-taken) after two mis-predictions as a standard 2-bit predictor. However, after one mis-prediction we remember the mis-prediction and switch back and forth after another mis-prediction. This is different from standard mis-predictor, once we change the prediction, we forget previous mis-predictions and we require 2 more mis-predictions before changing the prediction. Thus the predictor on the right (B) treats taken and not-taken paths equally.

b). Under what assumptions about the program behavior does A perform better than B. Likewise, under what assumptions about the program behavior does B perform better than A.
Since A favors predict taken branches, this method is well suited with applications containing nested loops (where the loops are executed many times, leading to taken branches).

```plaintext
for (..) {
    for (…) {
        …. 
        for (..) {}}
}
```

So this is better for scientific applications. Note that a standard 2-bit predictor also works for this type of a program since we do not change prediction from taken (while executing a loop) to not taken even at the end of the loop (which is not-taken branch) unless you mis-predict twice. However, the predictor B in the problem behaves somewhat differently from the standard 2-bit predictor since it switches back and forth between predict taken and not taken after one mis-prediction.

This predictor is also better if programs contain if..then..if..then.. type nesting (without else parts).

```plaintext
if (.) then
    if (..) then
        if(…) .....
B is better for cases where branches are equally likely to be taken or not taken.
    if (.) then … else
```

This may be the case with integer benchmarks without many loops.