VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY

HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY

DEPARTMENT OF ELECTRONICS

EE3043 – Computer Architecture

Semaster 241

Group	**:**	**23**	
Student	**:**	**Hoang Minh Chien**	**(2112932)**
		Doan Dinh Nam	**(2110368)**
		Giang Anh Đức	**(2011103)**
Teacher	**:**	**Dr. Tran Hoang Linh**	
TA	**:**	**Msc. Hai Cao Xuan**	

Ho Chi Minh City, November 17, 2024

PREFACE

This project focuses on the design and implementation of a **5-stage Pipelined RISC-V RV32I processor**, extending the previous single-cycle design. The project aims to address pipeline efficiency by incorporating **data forwarding**, **non-forwarding**, and **branch prediction** mechanisms to optimize performance and minimize hazards.

Key techniques include:

- **Non-Forwarding**: Baseline pipeline operation without data forwarding, evaluating performance under potential hazards.
- **Data Forwarding**: Enhancing pipeline efficiency by resolving data hazards with operand forwarding techniques.
- **Branch Prediction**: Implementing dynamic prediction strategies to reduce control hazards and improve instruction throughput.

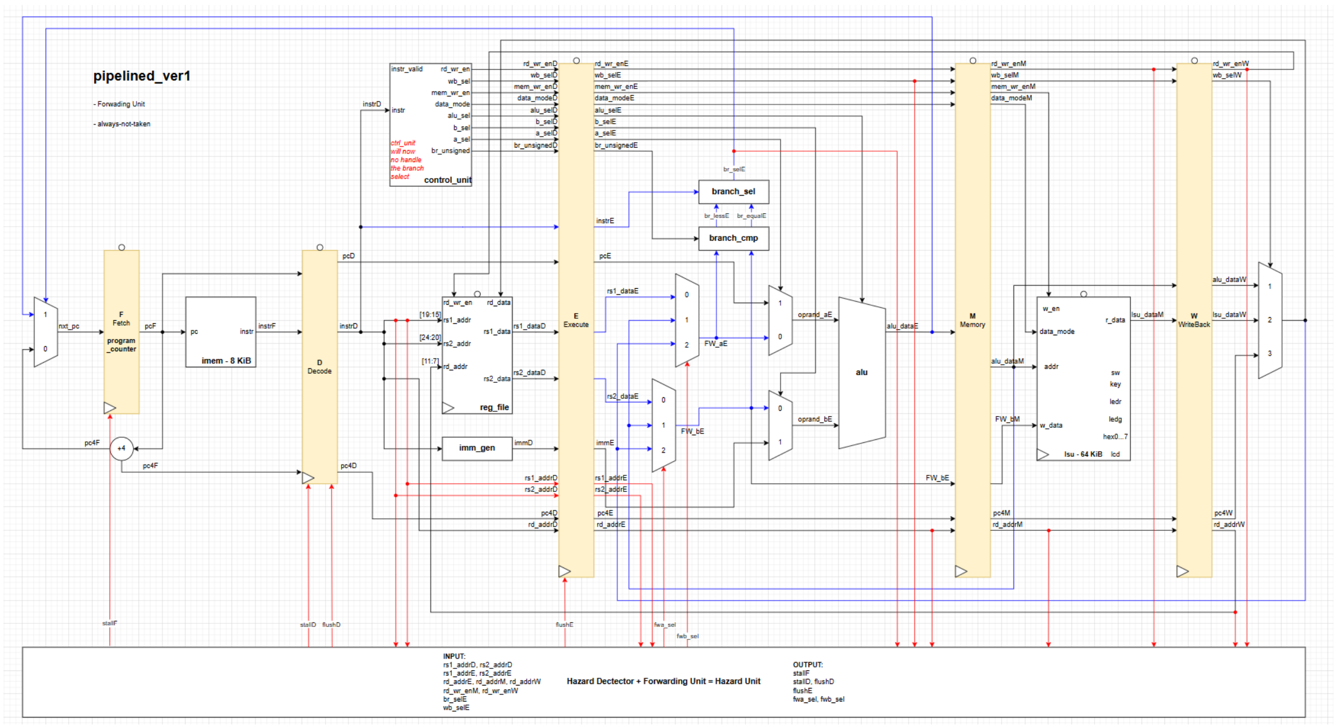
The processor's performance is evaluated by analyzing **Instruction Per Cycle (IPC)** under different benchmarks, providing a comprehensive comparison. Synthesis and implementation were carried out on the **DE2-Standard FPGA development board**, ensuring functional correctness and real-world applicability.

Through this project, we gained significant insights into the challenges of pipeline design, hazard resolution, and prediction models. This work sets a foundation for deeper exploration into performance optimization within modern CPU architectures.

CONTENTS

1. DESIGN STRATEGY

1.1. Baseline 5-Stage Pipelined: Nonforwarding & Forwarding



Basic Terminology

Before we dive into the design part, it's important to understand some basic terminology:

- Hazards occur when an instruction cannot execute in its expected cycle.
- There are two main types of hazards:
    1. Pipeline Hazards: This includes Data Hazards and Structure Hazards.
        - Data Hazard: Occurs when a planned instruction cannot be executed at the proper clock cycle because the required data for execution is not yet available.
    2. Control Hazard (or Branch Hazard): Occurs when the pipeline does not know the correct path to execute due to a branch instruction.

Design Strategy

The key idea of pipelining is to shorten the critical path by breaking it into multiple shorter paths. This allows us to increase the maximum operating frequency. However, the frequency increase is not linear to the number of pipeline stages, as it depends on the new critical path introduced. For a typical 5-stage pipeline, achieving a 2-3 times increase in frequency is considered good.

Now, let's move on to designing a 5-stage pipelined processor and identify the problems we encounter when dividing the design into five parts.

1. Data Hazard

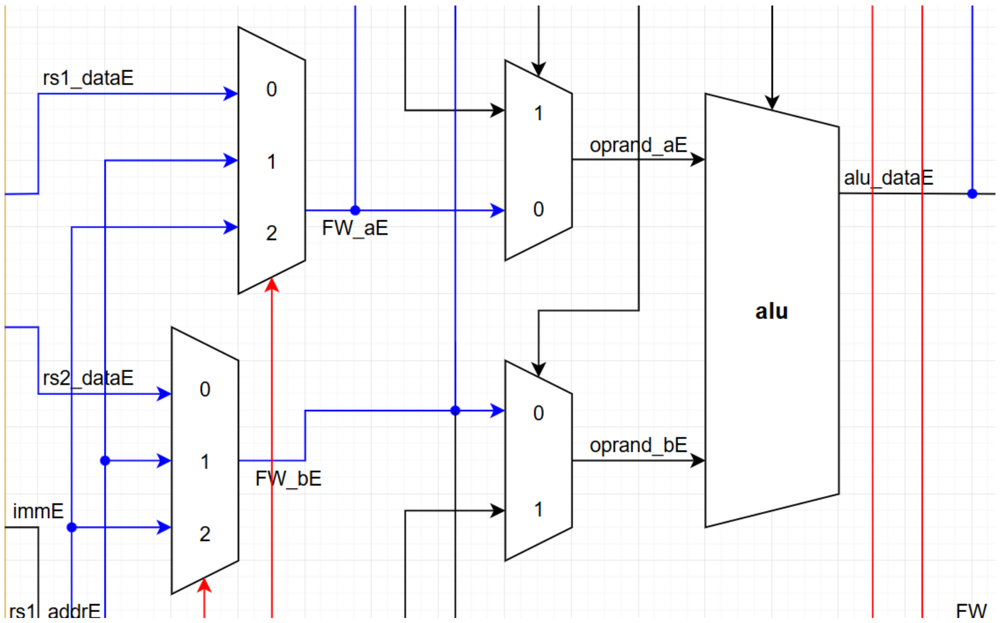
The foremost problem is Data Hazard, specifically RAW (Read After Write) hazards. There are two methods to deal with RAW hazards:

Non-Forwarding Design:

- In this approach, we insert NOPs until the result has been successfully written into the Register File.
- However, this approach dramatically slows down the processor due to the stalls caused by NOP insertion.

***Forwarding Design**:*

- To improve performance, we forward the result from the Memory Stage (M-stage) and Writeback Stage (WB-stage) directly to the Execute Stage (E-stage) without waiting for the data to be written to the Register File.
- To implement this, we insert two Forwarding MUXes before the two operand MUXes of the ALU.



When the RAW hazard detector detects a RAW hazard, it instructs the two Forwarding MUXes to select the data from the M-stage or WB-stage, instead of fetching rs1_data and rs2_data from the D-stage.

However, load instructions (e.g., lw) introduce an additional challenge because they have a two-cycle latency. For load-use hazards:

- We must stall the pipeline for one cycle when the RAW hazard detector identifies a load hazard. This happens when the destination register address in the E-stage matches the source register address in the D-stage.
- After stalling, the next instruction enters the E-stage and receives the forwarded data from the WB-stage of the previous lw instruction.

2. Control hazard

After resolving Data Hazards, we deal with Control Hazards caused by three types of branch instructions in the Base-Integer RISC-V ISA:

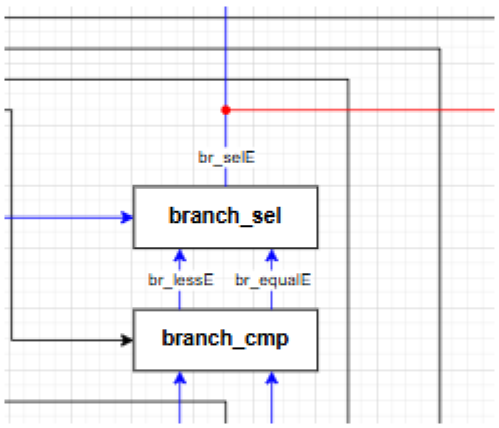
1. B-type Instructions (e.g., BEQ, BNE)
2. J-type Instructions (e.g., JAL)
3. I-type Instructions (e.g., JALR)

The solution for control hazards is to always fetch the next instruction until the correct branch direction is determined. If the branch is mispredicted, we flush the incorrectly fetched instructions and fetch the correct one (yes we wasted two cycle, that's called Branch Misprediction Penalty)

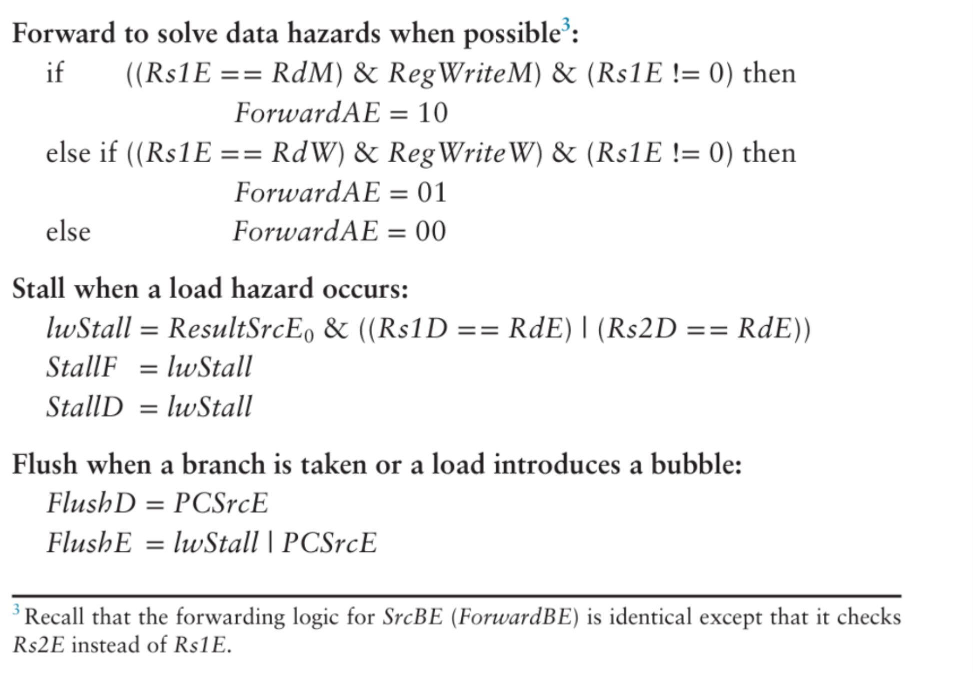
To fetch the correct instruction:

- We send the target address and a selection signal to the leftmost MUX (before the Program Counter).
- The correct branch direction is determined only when the instruction reaches the E-stage.

The branch_sel module at E-stage is used to control the selection input of the leftmost MUX, ensuring the correct target address is loaded into the Program Counter.



Summary of our Hazard Detector + Forwading Unit = Hazard Unit is:



Forward to solve data hazards when possible[3]:

```
if      $((Rs1E == RdM)$ & $RegWriteM)$ & $(Rs1E$ != $0)$ then
                $ForwardAE = 10$
else if $((Rs1E == RdW)$ & $RegWriteW)$ & $(Rs1E$ != $0)$ then
                $ForwardAE = 01$
else            $ForwardAE = 00$
```

Stall when a load hazard occurs:

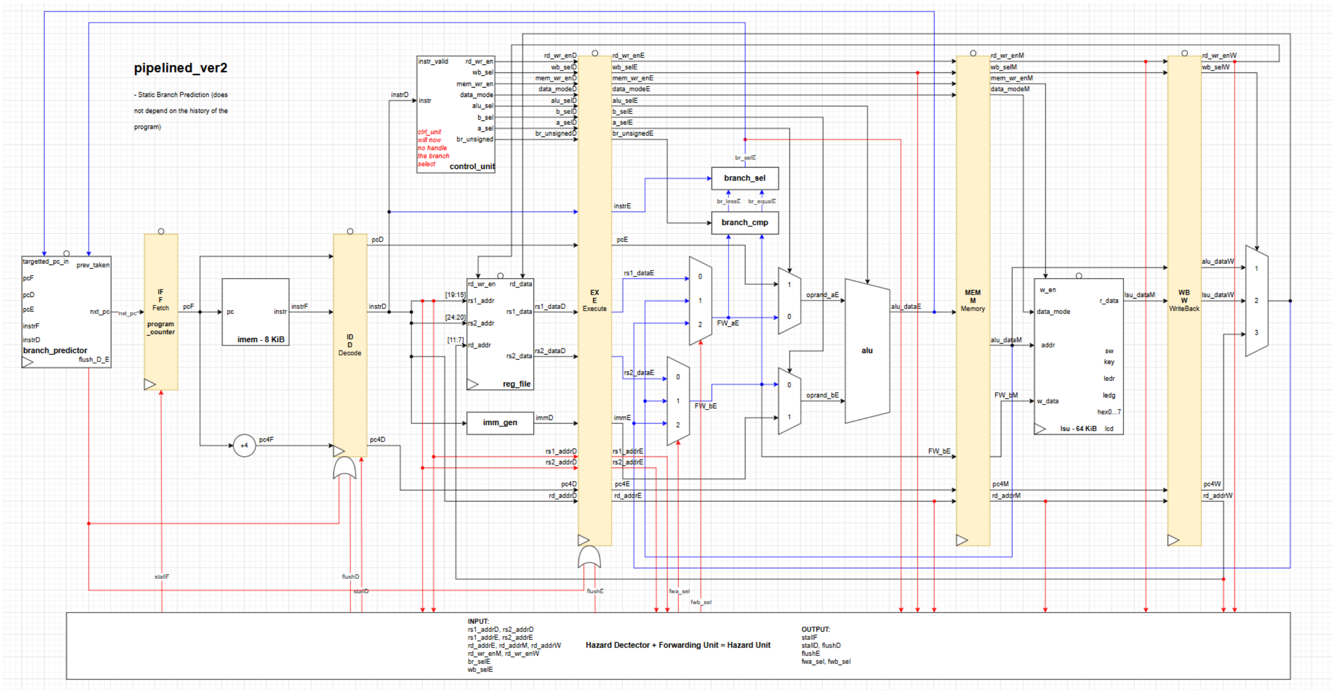
$lwStall = ResultSrcE_0$ & $((Rs1D == RdE) \mid (Rs2D == RdE))$
$StallF\ \ = lwStall$
$StallD\ \ = lwStall$

Flush when a branch is taken or a load introduces a bubble:

$FlushD = PCSrcE$
$FlushE\ = lwStall \mid PCSrcE$

[3] Recall that the forwarding logic for *SrcBE* (*ForwardBE*) is identical except that it checks *Rs2E* instead of *Rs1E*.

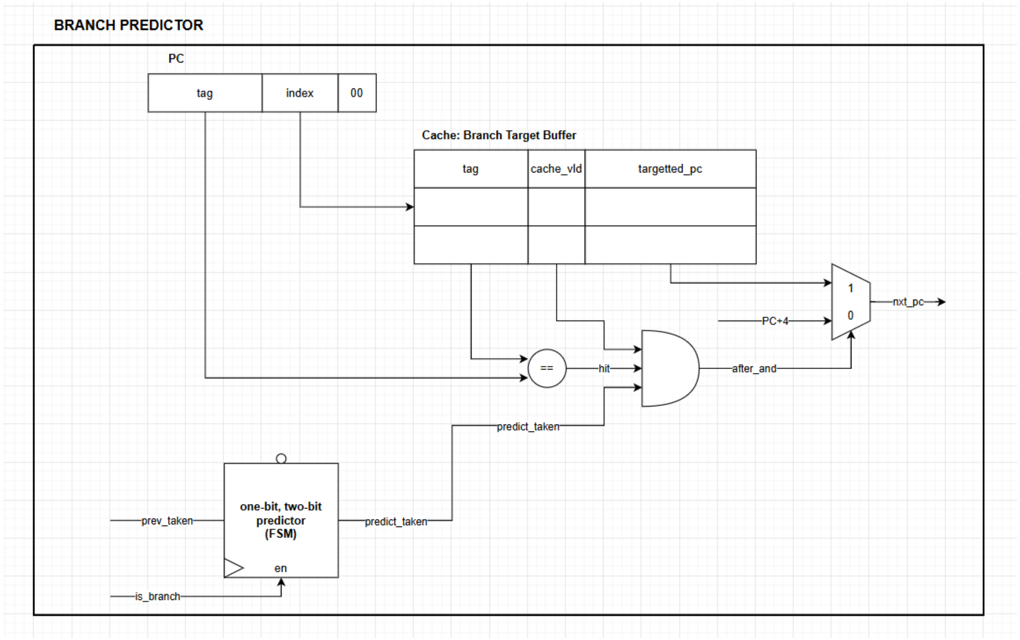
1.2. Branch Prediction

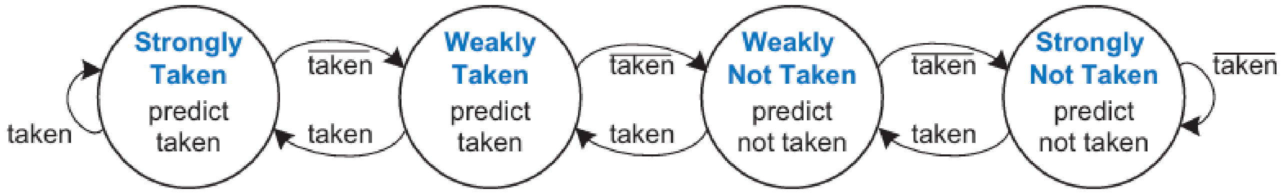


An ideal pipelined processor would have a CPI of 1. The branch misprediction penalty is a major reason for increased CPI. As pipelines get deeper, branches are resolved later in the pipeline. Thus, the branch misprediction penalty gets larger because all the instructions issued after the mispredicted branch must be flushed. To address this problem, most pipelined processors use a *branch predictor* to guess whether the branch should be taken.

Some branches occur at the beginning of a loop to check a condition and branch past the loop when the condition is no longer met (e.g., in *for* and *while* loops). Loops tend to execute many times, so these forward branches are usually not taken. Other branches occur when a program reaches the end of a loop and branches back to repeat the loop (e.g., in a *do/while* loop). Again, because loops tend to execute many times, these backward branches are usually taken. The simplest form of branch prediction checks the direction of the branch and predicts that backward branches are taken and forward branches are not. This is called *static branch prediction*, because it does not depend on the history of the program.

However, branches, especially forward branches, are difficult to predict without knowing more about the specific program. Therefore, most processors use dynamic branch predictors, which use the history of program execution to guess whether a branch should be taken. Dynamic branch predictors maintain a table of the last several hundred (or thousand) branch instructions that the processor has executed. The table, called a branch target buffer, includes the destination of the branch and a history of whether the branch was taken.





A one-bit dynamic branch predictor remembers whether the branch was taken the last time and predicts that it will do the same thing the next time. While the loop is repeating, it remembers that the *beq* was not taken last time and predicts that it should not be taken next time. This is a correct prediction until the last branch of the loop, when the branch does get taken. Unfortunately, if the loop is run again, the branch predictor remembers that the last branch was taken. Therefore, it incorrectly predicts that the branch should be taken when the loop is first run again. In summary, a 1-bit branch predictor mispredicts the first and last branches of a loop.

A two-bit dynamic branch predictor solves this problem by having four states: Strongly Taken, Weakly Taken, Weakly Not Taken, and Strongly Not Taken, as shown in *Figure 7.67*. When the loop is repeating, it enters the Strongly Not Taken state and predicts that the branch should not be taken next time. This is correct until the last branch of the loop, which is taken and moves the predictor to the Weakly Not Taken state. When the loop is first run again, the branch predictor correctly predicts that the branch should not be taken and reenters the Strongly Not Taken state. In summary, a two-bit branch predictor mispredicts only the last branch of a loop. It is called a two-bit branch predictor because it requires two bits to encode the four states.

The branch predictor operates in the Fetch stage of the pipeline so that it can determine which instruction to execute on the next cycle. When it predicts that the branch should be taken, the processor fetches the next instruction from the branch destination stored in the branch target buffer.
As one can imagine, branch predictors may be used to track even more history of the program to increase the accuracy of predictions. Good branch predictors achieve better than 90% accuracy on typical programs.

2. VERIFICATION STRATEGY

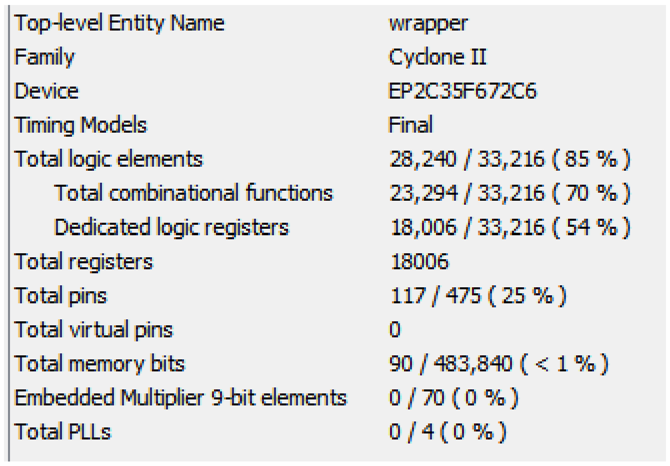
The verification strategy is nothing newer than our previous strategy.

We reuse the assembly program from previous single-cycle design, run it on the pipelined design to see whether it’s forwading, stall, flush correctly.

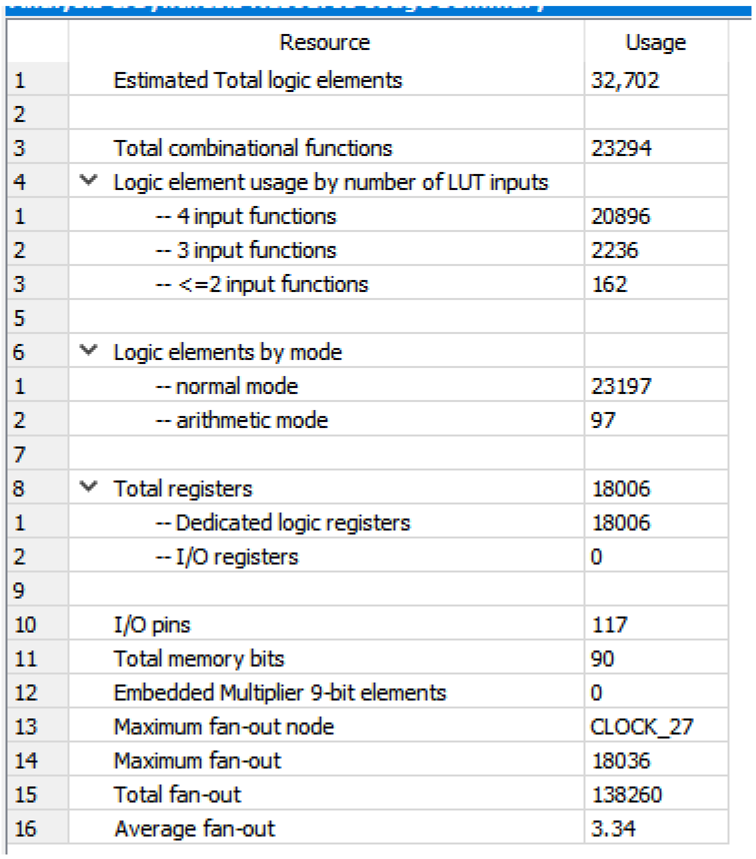
From now on, we call the assembly program is benchmark since it’s will be used to calculate our IPC.

3. RESULTS

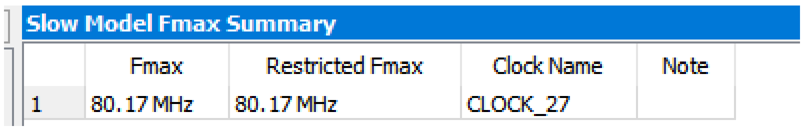
3.1. Synthesis



Top-level Entity Name	wrapper
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	28,240 / 33,216 (85 %)
Total combinational functions	23,294 / 33,216 (70 %)
Dedicated logic registers	18,006 / 33,216 (54 %)
Total registers	18006
Total pins	117 / 475 (25 %)
Total virtual pins	0
Total memory bits	90 / 483,840 (< 1 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)



	Resource	Usage
1	Estimated Total logic elements	32,702
2		
3	Total combinational functions	23294
4	Logic element usage by number of LUT inputs	
1	-- 4 input functions	20896
2	-- 3 input functions	2236
3	-- <=2 input functions	162
5		
6	Logic elements by mode	
1	-- normal mode	23197
2	-- arithmetic mode	97
7		
8	Total registers	18006
1	-- Dedicated logic registers	18006
2	-- I/O registers	0
9		
10	I/O pins	117
11	Total memory bits	90
12	Embedded Multiplier 9-bit elements	0
13	Maximum fan-out node	CLOCK_27
14	Maximum fan-out	18036
15	Total fan-out	138260
16	Average fan-out	3.34



Slow Model Fmax Summary

	Fmax	Restricted Fmax	Clock Name	Note
1	80.17 MHz	80.17 MHz	CLOCK_27	

3.2. Performance (IPC)

As we present in the presentation day.

The key of how to compure IPC is run the benchmark on both single-cycle and pipelined design.

Since the Single Cycle design always have IPC = 1, then we choose a reference point where the program release the output, we count total cycle, then the total valid instructions ic 1*total cycle.

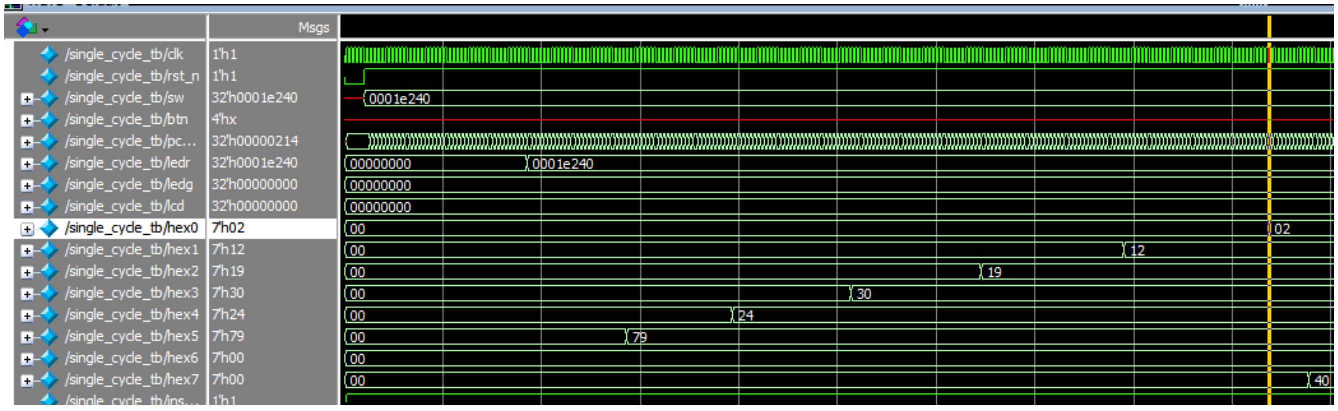
We save that instruction.

Next we run the same benchmarks on piplined, we determinted the reference point (excatly the same point at which we get the same result with the single cycle simulation). Then record the total cycle.

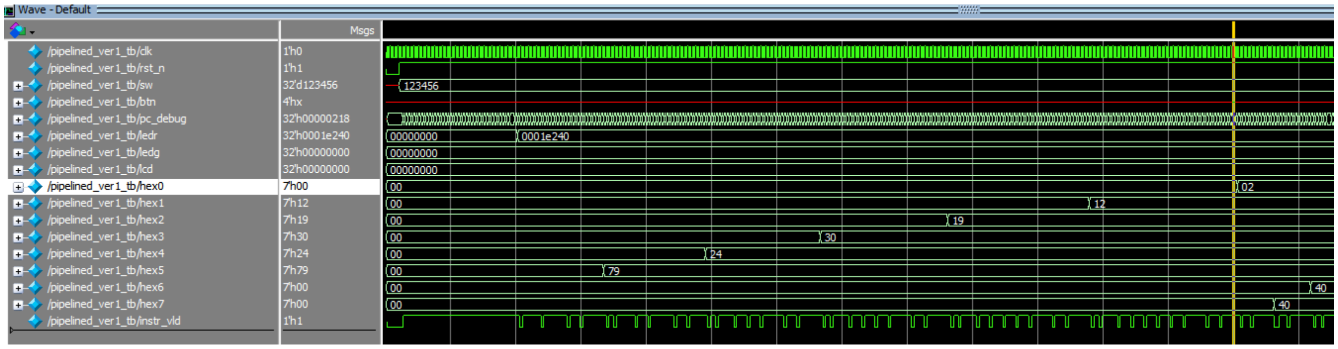
Finally, we simply divide the total valid instruction (we have this at singlecycle simulatin) from total cycle.

Example:

With the HEX2SWITCH program, the reference point is the point where all 6 hex was computed:



```
                                   Msgs
/single_cycle_tb/clk               1'h1
/single_cycle_tb/rst_n             1'h1
/single_cycle_tb/sw                32'h0001e240
/single_cycle_tb/btn               4'hx
/single_cycle_tb/pc...             32'h00000214
/single_cycle_tb/ledr              32'h0001e240
/single_cycle_tb/ledg              32'h00000000
/single_cycle_tb/lcd               32'h00000000
/single_cycle_tb/hex0              7'h02
/single_cycle_tb/hex1              7'h12
/single_cycle_tb/hex2              7'h19
/single_cycle_tb/hex3              7'h30
/single_cycle_tb/hex4              7'h24
/single_cycle_tb/hex5              7'h79
/single_cycle_tb/hex6              7'h00
/single_cycle_tb/hex7              7'h00
/single_cycle_tb/ins...            1'h1
```



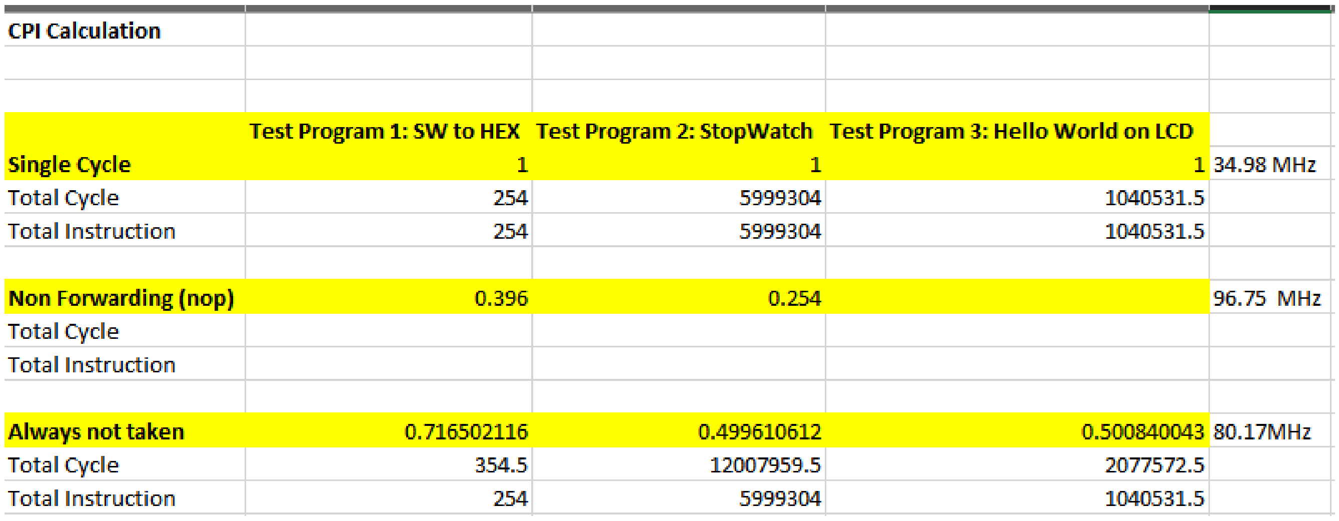
```
Wave - Default
                                   Msgs
/pipelined_ver1_tb/clk             1'h0
/pipelined_ver1_tb/rst_n           1'h1
/pipelined_ver1_tb/sw              32'd123456
/pipelined_ver1_tb/btn             4'hx
/pipelined_ver1_tb/pc_debug        32'h00000218
/pipelined_ver1_tb/ledr            32'h0001e240
/pipelined_ver1_tb/ledg            32'h00000000
/pipelined_ver1_tb/lcd             32'h00000000
/pipelined_ver1_tb/hex0            7'h00
/pipelined_ver1_tb/hex1            7'h12
/pipelined_ver1_tb/hex2            7'h19
/pipelined_ver1_tb/hex3            7'h30
/pipelined_ver1_tb/hex4            7'h24
/pipelined_ver1_tb/hex5            7'h79
/pipelined_ver1_tb/hex6            7'h00
/pipelined_ver1_tb/hex7            7'h00
/pipelined_ver1_tb/instr_vld       1'h1
```

With the STOPWATCH program, the reference point is the point where all centisecond numer is :

the simulation time take too long .... so the result will not be captrued here =))

With the LCDHELLOWORLD program, the reference point is the point where io_lcd push out 20 command to LCD display:

the simulation time take too long .... so the result will not be captrued here =))



CPI Calculation

	Test Program 1: SW to HEX	Test Program 2: StopWatch	Test Program 3: Hello World on LCD	
Single Cycle	1	1	1	34.98 MHz
Total Cycle	254	5999304	1040531.5	
Total Instruction	254	5999304	1040531.5	
Non Forwarding (nop)	0.396	0.254		96.75 MHz
Total Cycle				
Total Instruction				
Always not taken	0.716502116	0.499610612	0.500840043	80.17MHz
Total Cycle	354.5	12007959.5	2077572.5	
Total Instruction	254	5999304	1040531.5	

The result showed on this table. As we can see, as the simulation long (6 millions valid instructiuons), the IPC reach 0.5 since my assembly have a bunch of delay loop.

Another clever way to compute IPC is looking at the valid and invalid instruction at instr_vld pin of piplined, the write a script to counter valid, invalid instructions. Then, do math.

4. FPGA IMPLEMENTATION

Everything was the same with our previous Single-Cycle RISC-V report.