**ELG5195 Project**
Report

# Pipelined Processor

Submitted by

| SID | Names of Students |
|---|---|
| | |
| | |

Under the guidance of
**Dr. Voicu Groza**

uOttawa

Winter Semester 2014

**Abstract**

Pipelined processor using the idea of splitting one instruction into a sequence of dependent steps to increase the speed of the processor. In our project, we implemented a pipelined processor which can execute some of the 8-bit AVR instruction set using VHDL. It consists of 3 stages: opcode fetching, opcode decoding and executing, and overcomes 3 well known hazards: structural, control and data hazards. It has 2 features that are rarely found in general pipelined processor. First, it uses dual port memory for program memory, which means it can be read and written by two modules at the same time. Second, we put the register file in the executing stage, thus avoiding data structure.

The report is organized as follows. Chapter 1 introduces the basic idea about pipelined processor including the motivation and the problems which should be solved and how to solve them. A brief review of previous work is also presented. In Chapter 2, the design is described in details. Finally in Chapter 3, the simulation and synthesis results are presented.

# Contents

# List of Figures

# Chapter 1

# Theoretical Part

## 1.1 Why Pipelined Processor

Applying pipeline model in the design of central processing units has following advantages. The cycle time of the processor is reduced and the instruction throughput is increased while applying pipeline model. Pipelining does not reduce the time it takes to complete an instruction, instead it increases the number of instructions that can be processed simultaneously and reduces the delay between completed instructions. The CPU arithmetic logic unit can be designed faster if pipelining is used. Besides, pipelining increases performance over an un-pipelined core by a factor of the number of stages and the code is ideal for pipeline execution. Briefly state, pipeline can make complex operations faster and more economically.
Assume the circuit shown below, consists of 3 combinational functions F1, F2 and F3, one flip-fop after F3. Let the 3 combinational functions have their own propagation delays T1, T2 and T3 respectively, then the total delay of the combinational is T=T1+T2+T3. Hence, the maximum frequency of this circuit f(max)=1/(T1+T2+T3).

Instead of the original circuit, we apply pipeline model, added flip-flops between the

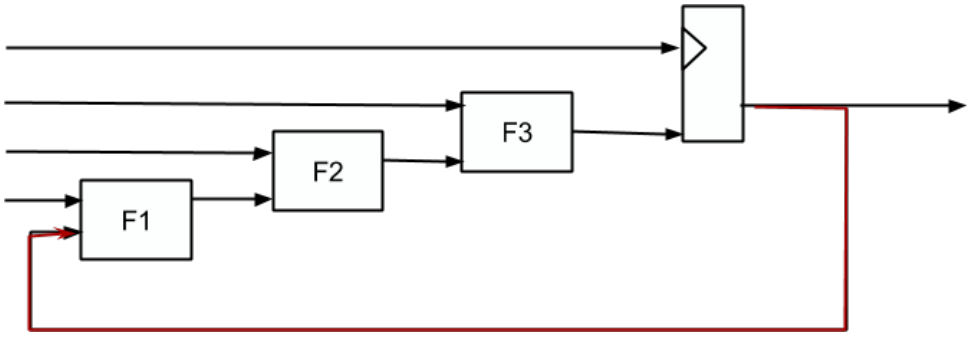



Figure 1.1: A digital sequential circuit without pipeline.

stages as shown below, while pipelining is a technique to increase the delay of a combinational circuit slightly but allows different components of the logic at the same time. Then, the delay for the slowest path is equal to the maximum among T1, T2 and T3. Hence, this new circuit with pipelining can be clocked with frequency: f(max)=1/max(T1,T2,T3).

Compared the different maximum frequency of processing in the two logic circuit above,

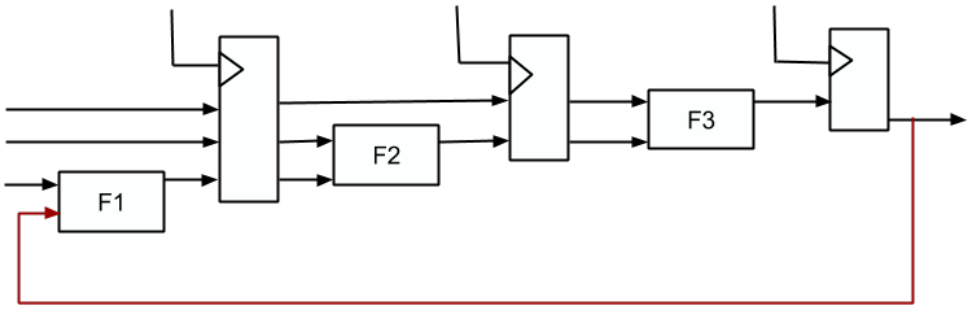



Figure 1.2: A digital sequential circuit with pipeline.

we can tell the frequency of the pipelining circuit is higher than which in the original circuit clearly. If the functions F1, F2 and F3 had equal propagation delays, then the maximum frequency of the new circuit would have tripled compared to the old circuit. The reason for the improved throughput is that the different stages of a pipeline work in parallel while without pipelining the entire logic would be occupied by a single operation.

## 1.2 Hazards in Pipelined Processor

Pipelining can efficiently increase the performance of a processor by overlapping execution of instructions. But the efficiency of the pipelining depends upon, how the problem encountered during the implementation of pipelining is handled. These problems are known as HAZARDS.
Types of Hazards:

1. Structural Hazards (Resource Bound)
2. Control Hazards (Pipelining Bubbles)
3. Data Hazards (Data Dependencies)

### 1.2.1 Structure Hazards

During the pipelining, the overlapped execution of instructions requires pipelining of functional units and duplication of resources to allow all possible combinations of instructions in the pipeline. If some combination of instructions cannot be accommodated because of a resource conflict, the machine is said to have a structural hazard. This type of hazards occurs when two activities require the same resource simultaneously.
Common instances of structural hazards arise when:
1) Some functional unit is not fully pipelined then a sequence of instructions using that un-pipelined unit cannot proceed at the rate of one per clock cycle.
2) Some resource has not been duplicated enough to allow all combinations of instructions in the pipeline to execute.

Example:
If machine has shared a single-memory pipeline for data and instructions. As a result, when an instruction contains a data-memory reference (load-MEM), it will conflict with the instruction reference for a later instruction (instr 3-IF):

| Clock cycle number | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Instr** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | | |
| Load | IF | ID | EX | MEM* | WB | | | | | |
| Instr 1 | | IF | ID | EX | MEM | WB | | | | |
| Instr 2 | | | IF | ID | EX | MEM | WB | | | |
| Instr 3 | | | | IF* | ID | EX | MEM | WB | | |

Figure 1.3: Structure Hazards (1)

To resolve this, we stall the pipeline for one clock cycle when a data-memory access occurs. The effect of the stall is actually to occupy the resources for that instruction slot. The following table shows how the stalls are actually implemented.

Instruction 1 assumed not to be data-memory reference (load or store), otherwise In-

| Clock cycle number | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Instr** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | |
| Load | IF | ID | EX | MEM | WB | | | | | |
| Instr 1 | | IF | ID | EX | MEM | WB | | | | |
| Instr 2 | | | IF | ID | EX | MEM | WB | | | |
| Instr 3 | | | | stall | IF | ID | EX | MEM | WB | |

Figure 1.4: Structure Hazards (2)

struction 3 cannot start execution because of structural hazard.
We know that introduction of stall reduces the performance of the system but there are following reasons for allowing structural hazard while designing the system:
To reduce cost: For example, machines that support both an instruction and a cache access every cycle (to prevent the structural hazard of the above example) require at least twice as much total memory. To reduce the latency of the unit: The shorter latency comes from the lack of pipeline registers that introduce overhead.
Structural hazards is solved by using dual port memory, which can be accessed by two modules at the same time in our project.

### 1.2.2 Control Hazards

This type of hazard is caused by uncertainty of execution path, branch taken or not taken. It is a hazard that arises when an attempt is made to make a decision before condition is evaluated. It results when we branch to a new location in the program, invalidating everything we have loaded in our pipeline. Control hazard can cause a greater performance loss for DLX pipeline than data hazards. When a branch is executed, it may or may not change the PC (program counter) to something other than its current value plus 4. If a

branch changes the PC to its target address, it is a taken branch; if it falls through, it is not taken. If instruction i is a taken branch, then the PC is normally not changed until the end of MEM stage, after the completion of the address calculation and comparison.
Methods to Deal with Control Hazard:
Pipeline stall until branch target known.
Continue fetching instructions as if we wont take the branch but then invalidating the instruction if we do take the branch.
- Always fetch the branch target After all, most branches are taken.
- Precompute the target if architecture support (DLX doesnt support it).
Delayed Branch: Perform instruction scheduling into branch delay slots (instruction after a branch)
Always execute instructions following a branch regardless of whether branch taken or not taken.
The simplest method of dealing with branches is to stall the pipeline as soon as the branch is detected until we reach the MEM stage, which determines the new PC. The pipeline behavior looks like:
The stall does not occur until after ID stage (where we know that the instruction is a

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Branch | IF | **ID** | EX | **MEM** | WB | | | | | |
| Branch successor | | **IF**(stall) | ***stall*** | ***stall*** | IF | ID | EX | MEM | WB | |
| Branch successor+1 | | | | | | IF | ID | EX | MEM | WB |

Figure 1.5: Contro Hazards (1)

branch). This control hazard stall must be implemented differently from a data hazard, since the IF cycle of the instruction following the branch must be repeated as soon as we know the branch outcome. Thus, the first IF cycle is essentially a stall (because it never performs useful work), which comes to total 3 stalls. Three clock cycles wasted for every branch is a significant loss.
With a 30% branch frequency and an ideal CPI of 1, the machine with branch stalls achieves only half the ideal speedup from pipelining.
The number of clock cycles can be reduced by two steps:

- Find out whether the branch is taken or not taken earlier in the pipeline;
- Compute the taken PC (i.e., the address of the branch target) earlier.

The Branch delay slot scheduling method helps reducing the branch penalty.
There are three modes of scheduling the delay slot in the branching instructions:

- Scheduling before the branch target instruction.
- Scheduling from the target instruction.

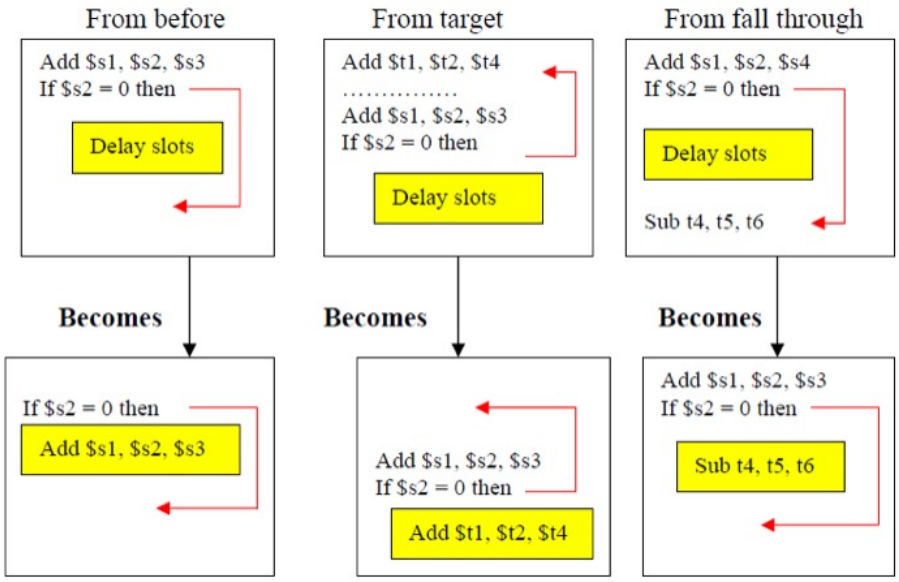



Figure 1.6: Contro Hazards (2)

- Scheduling from fall through instruction.

In our design, a skip signal is driven by the executing stage, which connects to the opcode fetching stage and decoding stage. When a branch needs to be implemented, the skip signal will equals to 1, causes the opcode decoding stage to generate empty control signal for the executing stage.

### 1.2.3 Data Hazards

Data hazards occur when the pipeline changes the order of read/write accesses to operands so that the order differs from the order seen by sequentially executing instructions on the unpipelined machine.
Data hazards are also known as data dependency. Data dependency is the condition in which the outcome of the current operation is dependent on the outcome of a previous instruction that has not yet been executed to completion because of the effect of the pipeline.
Data hazards arise because of the need to preserve the order of the execution of instructions. The following example shows the data hazards: Data Hazards will not occur in

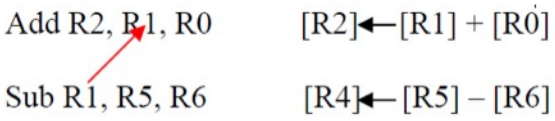



Figure 1.7: Data Hazards

our design because the register file is only accessed by the executing stage, not by the opcode decoding stage.

## 1.3 Literature Review

Hazards in pipelined processor is not a new field, many research have been done since 20 years ago. Techniques such as CATHEDRA II[8], HAL[12], PLS[10] are developed for DSP applications. They avoid pipeline hazards by limiting the degree of pipelining. Another method called "Snapshot" is developed by Cloutier for pipelined ISP synthesis[7]. His approach is subject to the same limitation of MAL. A synthesis system Piper for pipelined processor was proposed by Ing-Jer Huang etc[9], which further improves the performance by simultaneously generating pipelined micro-architectures and their compiler backends. Higher throughput can be obtained by synthesizing pipelined designs with initiation latencies less than MALs. This approach has the advantage over ASPD's in that the compiler backends can make use of the cycles which would otherwise be stalled in ASPD's designs

As we try to avoid the hazards in pipelined processor, we have two choice, one is to choose a complicated way which require no stalls or a simple way which require less hardware. The choice is not easy, we need to balance between many considerations. So a method to balance between hardware cost and the cost of degraded performance is proposed by Casavant[6]. But this method only considers structural hazards. Data hazards and control hazards should be added to the cost function. Recently, multithreading has been studied, which is distinguished from multiprocessing systems (such as multi-core systems) in that the threads have to share the resources of a single core: the computing units, the CPU caches and the translation lookaside buffer. Where multiprocessing systems include multiple complete processing units, multithreading aims to increase utilization of a single core by using thread-level as well as instruction-level parallelism. As the two techniques are complementary, they are sometimes combined in systems with multiple multithreading CPUs and in CPUs with multiple multithreading cores.

Manjikian etc proposed a design which implement the multithreading in a pipelined 32-bit processor[11]. Synthesis results obtained using Altera Quartus II for a Stratix logic chip indicate an overhead of 21% for multithreading in terms of logic elements, with the largest contribution from multiple program counters and the associated logic to multiplex their outputs and set their contents. There is no additional overhead related to embedded memory blocks because previously-unused storage is allocated to distinct register files for multiple threads.

# References


[1] http://websrv.cs.fsu.edu/ tyson/cda5155/refs.html.

[2] http://www.cs.caltech.edu/courses/cs184/winter2001/slides/day5_2up.pdf.

[3] http://www.cs.iastate.edu/p̃rabhu/tutorial/pipeline/compsched.html.

[4] http://www.cs.ucsd.edu/classes/wi99/cse141_b/lectures.html.

[5] http://www.mmdb.ece.ucsb.edu/ẽce154/lecture5.pdf.

[6] A.E. Casavant. Balancing structural hazards and hardware cost of pipelined processors. In *European Design and Test Conference, 1995. ED TC 1995, Proceedings.*, pages 562–566, Mar 1995.

[7] Richard J. Cloutier and Donald E. Thomas. Synthesis of pipelined instruction set processors. In *Proceedings of the 30th International Design Automation Conference*, DAC '93, pages 583–588, New York, NY, USA, 1993. ACM.

[8] G. Goossens, J. Rabaey, J. Vandewalle, and H. De Man. An efficient microcode compiler for application specific dsp processors. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 9(9):925–937, Sep 1990.

[9] Ing-Jer Huang and A.M. Despain. High level synthesis of pipelined instruction set processors and back-end compilers. In *Design Automation Conference, 1992. Proceedings., 29th ACM/IEEE*, pages 135–140, Jun 1992.

[10] Cheng-Tsung Hwang, Yu-Chin Hsu, and Youn-Long Lin. Scheduling for functional pipelining and loop winding. In *Design Automation Conference, 1991. 28th ACM/IEEE*, pages 764–769, June 1991.

[11] N. Manjikian. Implementation of hardware multithreading in a pipelined processor. In *Circuits and Systems, 2006 IEEE North-East Workshop on*, pages 145–148, June 2006.

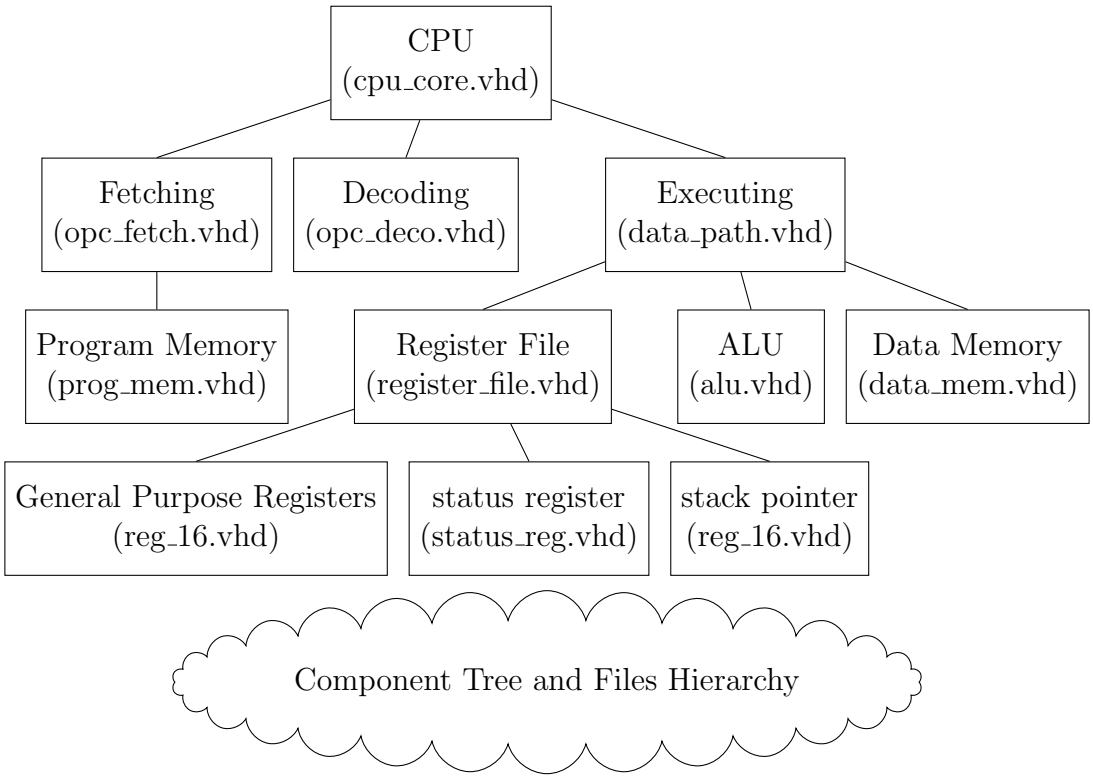
[12] P.G. Paulin and J.P. Knight. Force-directed scheduling for the behavioral synthesis of asics. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 8(6):661–679, Jun 1989.

[13] Juergen Sauermann. http://opencores.org/usercontent,doc,126270725.

# Chapter 2

# Design

Our design is based on the work of Juergen Sauermann[13]. All source codes and schematics have published on https://github.com/d8660091/ELG5195Project. The following figure shows all components and their corresponding VHDL source file in our design. The CPU is divided to 3 stages: opcode fetching, decoding and datapath(executing).



## 2.1 Components Overview

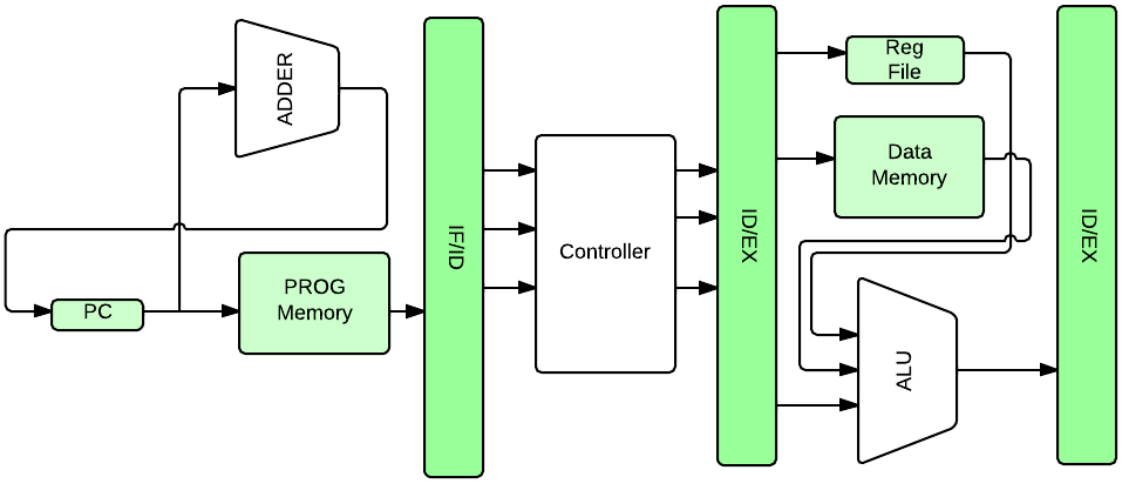



Figure 2.1: Simplified structure of the pipelined processor

A simplified structure of our pipelined processor is described in figure 2.1. This image demonstrates the basic working principle of the processor. Firstly, the instruction fetching stage fetching an opcode according to the value of the PC(Program Counter), then output this opcode to the output port for using by the next stage, opcode decoding. Then add PC by 1,2 or k, depends on the instruction length or whether the previous opcode is a branch instruction. Secondly, the opcode decoding stage will convert the opcode to control signal, which will control the behaviour of the datapath, i.e. executing stage. Finally, the executing stage will write the value to memory when next clock is coming.
It should noticed here that the register file is located in the executing stage, unlike other general pipelined processor, which means the data hazards will not occur here. Because the register file is accessed only by the executing stage, instead of being accessed by both the decoding and executing stage.

## 2.2 Opcode Fetch

The opcode fetch stage is the simplest stage in the pipeline. It is the stage that put life into the CPU core by generating a sequence of opcodes that are then decoded and executed. The opcode fetch stage is sometimes called the sequencer of the CPU.

```
entity opc_fetch is
    port ( I_CLK      : in  std_logic; --Clock Input.

           I_CLR      : in  std_logic; --Clear Signal.
```

```
            --Program Counter input indicator.
            I_LOAD_PC  : in  std_logic;
            --Program Counter input value.
            I_NEW_PC   : in  std_logic_vector(15 downto 0);
            --Address port for executing stage.
            I_PM_ADR   : in  std_logic_vector(11 downto 0);
            --Skip indicator drive by executing stage.
            I_SKIP     : in  std_logic;

            --Opcode output.
            Q_OPC      : out std_logic_vector(31 downto 0);
            --Current Program Counter value.
            Q_PC       : out std_logic_vector(15 downto 0);
            --Output for reading data from executing stage.
            Q_PM_DOUT  : out std_logic_vector( 7 downto 0);
            --2 clocks instruction indicator.
            Q_T0       : out std_logic);
end opc_fetch;
```

The PC is updated on every clock with its next value. The T0 output is cleared when the WAIT signal is raised. This causes the T0 output to be '1' on the first cycle of a 2 cycle instruction and '0' on the second cycle.
I_LOAD_PC equals to 1 when instruction like JMP or CALL is encountered. And the 16 bits input I_NEW_PC is the location of next instruction.
I_PM_ADR is another address port for the program memory (the other address port is PC), which means that the program memory can be accessed by 2 stage: opcode fetching and executing at the same time. Thus the structural hazards are avoided. Q_PM_DOUT is the output port for this address.
I_SKIP signal is driven by the executing stage when instruction like JMP or CALL is encountered, which means the next instruction (PC+1 or 2) should be skipped. The output port of Q_OPC will change to X"0000" immediately without waiting for next clock, which also allows the flushing of instruction in the decoding stage.
Q_T0 equals to 1 at first clock and 0 at the second clock when 2 clocks instruction encountered. Otherwise, it always equals to 1.

### 2.2.1 Program Memory

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

-- content of program memory.
use work.prog_mem_content.all;

entity prog_mem is
  generic ( INIT_00 : std_logic_vector (15 downto 0) := p_00;
            INIT_01 : std_logic_vector (15 downto 0) := p_01;
```

```
            INIT_02 : std_logic_vector (15 downto 0) := p_02;
            INIT_03 : std_logic_vector (15 downto 0) := p_03;
            INIT_04 : std_logic_vector (15 downto 0) := p_04;
            INIT_05 : std_logic_vector (15 downto 0) := p_05;
            INIT_06 : std_logic_vector (15 downto 0) := p_06;
            INIT_07 : std_logic_vector (15 downto 0) := p_07;
            INIT_08 : std_logic_vector (15 downto 0) := p_08;
            INIT_09 : std_logic_vector (15 downto 0) := p_09;
            INIT_0A : std_logic_vector (15 downto 0) := p_0A );
  port ( I_CLK      : in  std_logic;

         I_WAIT     : in  std_logic;
         I_PC       : in  std_logic_vector(15 downto 0); -- word address
         I_PM_ADR   : in  std_logic_vector(11 downto 0); -- byte address

         Q_OPC      : out std_logic_vector(31 downto 0);
         Q_PC       : out std_logic_vector(15 downto 0);
         Q_PM_DOUT  : out std_logic_vector( 7 downto 0) );
end prog_mem;

architecture behavioral of prog_mem is
  type memory_array is
    array (integer range 0 to 1000) of std_logic_vector(15 downto 0);
    signal L_memory : memory_array := (
                                       0 => INIT_00,
                                       1 => INIT_01,
                                       2 => INIT_02,
                                       3 => INIT_03,
                                       4 => INIT_04,
                                       5 => INIT_05,
                                       6 => INIT_06,
                                       7 => INIT_07,
                                       8 => INIT_08,
                                       9 => INIT_09,
                                       10 => INIT_0A,
                                       others => (others => '0'));
begin
  pc0: process(I_CLK)
  begin
    if (rising_edge(I_CLK)) then
      if (I_WAIT='0') then
        Q_PC <= I_PC;
        Q_OPC(15 downto 0) <= L_memory(to_integer(unsigned(I_PC)));
        Q_OPC(31 downto 16) <= L_memory(to_integer(unsigned(I_PC))+1);
      end if;
      if (I_PM_ADR(0)='0') then
        Q_PM_DOUT <= L_memory(to_integer(unsigned(I_PM_ADR)))(7 downto 0);
      else
```

```
        Q_PM_DOUT <= L_memory(to_integer(unsigned(I_PM_ADR)))(15 downto 8);
      end if;
    end if;
  end process;
end behavioral;
```

The program memory is a dual port memory. This means that two different memory locations can be read or written at the same time. We don't write to the program memory, be we would like to read two addresses at the same time. The reason are the LPM (load program memory) instructions. These instructions read from the program memory while the program memory is fetching the next instructions. In a way these instructions violate the Harvard architecture, but on the other hand they are extremely useful for string constants in C.
Rather than initializing the (typically smaller) data memory with these constants, one can leave them in program memory and access them using LPM instructions, Without a dual port memory, we would have needed to stop the pipeline during the execution of LPM instructions. Use of dual port memory avoids this additional complexity.
The second port used for LPM instructions consists of the address input PM_ADR and the data output PM_DOUT. PM_ADR is a 12-bit byte address (and consequently PM_DOUT is an 8-bit output. In contrast, the other port uses an 11-bit word address.
The other signals of the program memory belong to the first port which is used for opcode fetches.

### Memory Instantiation

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;


package prog_mem_content is
constant p_00 : std_logic_vector := X"E080"; -- LOAD: R24 <- 0
constant p_01 : std_logic_vector := X"9601"; -- ADIW: R24 <- R24+1
constant p_02 : std_logic_vector := X"2FA8"; -- MOV: R26 <- R24
constant p_03 : std_logic_vector := X"50AA"; -- SUBI: R26 <- 26-10
constant p_04 : std_logic_vector := X"940C"; -- JMP: to start 32 bits
constant p_05 : std_logic_vector := X"0001";
constant p_06 : std_logic_vector := X"FFFF";
constant p_07 : std_logic_vector := X"FFFF";
constant p_08 : std_logic_vector := X"FFFF";
constant p_09 : std_logic_vector := X"FFFF";
constant p_0A : std_logic_vector := X"FFFF";
end prog_mem_content;
```

The content of the program memory is put in a separated file instead the same file as the entity, named prog_mem_content.vhd for flexibility. Here, a simple program is presented, it including 5 different instructions. Which use the general register R24 and R26 as counters, and R26 equals to R24-10.

## 2.3 Opcode Decode

```
entity opc_deco is
    port ( I_CLK      : in  std_logic;

           I_OPC      : in  std_logic_vector(31 downto 0);
           I_PC       : in  std_logic_vector(15 downto 0);
           I_T0       : in  std_logic;

           Q_ALU_OP   : out std_logic_vector( 4 downto 0);
           Q_AMOD     : out std_logic_vector( 5 downto 0);
           Q_BIT      : out std_logic_vector( 3 downto 0);
           Q_DDDDD    : out std_logic_vector( 4 downto 0);
           Q_IMM      : out std_logic_vector(15 downto 0);
           Q_JADR     : out std_logic_vector(15 downto 0);
           Q_OPC      : out std_logic_vector(15 downto 0);
           Q_PC       : out std_logic_vector(15 downto 0);
           Q_PC_OP    : out std_logic_vector( 2 downto 0);
           Q_PMS      : out std_logic; -- program memory select
           Q_RD_M     : out std_logic;
           Q_RRRRR    : out std_logic_vector( 4 downto 0);
           Q_RSEL     : out std_logic_vector( 1 downto 0);
           Q_WE_01    : out std_logic;
           Q_WE_D     : out std_logic_vector( 1 downto 0);
           Q_WE_F     : out std_logic;
           Q_WE_M     : out std_logic_vector( 1 downto 0);
           Q_WE_XYZS  : out std_logic);
end opc_deco;
```

### 2.3.1 Inputs and Outputs

- CLK is the clock signal. The opcode decoder is a pure pipeline stage so that no internal state is kept between clock cycles. The output of the opcode decoder is a pure function of its inputs.
- OPC is the opcode being decoded. PC is the program counter (the address in the program memory from which OPC was fetched).
- T0 is '1' in the first cycle of the execution of the opcode. This allows for output signals of two-cycle instructions that are different in the first and the second cycle.
- ALU_OP defines which particular ALU operation (like ADD, ADC, AND, ...) the ALU shall perform.
- AMOD defines which addressing mode (like absolute, Z+, -SP, etc.)shall be used for data memory accesses.
- BIT is a bit value (0 or 1) and a bit number used in bit instructions.

- DDDDD defines the destination register or register pair (if any) for storing the result of an operation. It also defines the first source register or register pair of a dyadic instructions.
- IMM defines an immediate value or branch address that is computed from the opcode.
- JADR is a branch address.
- OPC is the opcode being decoded, or 0 if the opcode was invalidated by means of SKIP.
- PC is the PC from which OPC was fetched.
- PC_OP defines an operation to be performed on the PC (such as branching).
- PMS is set when the address defined by AMOD is a program memory address rather than a data memory address.
- RD_M is set for reads from the data memory.
- RRRRR defines the second register or register pair of a dyadic instruction.
- RSEL selects the source of the second operand in the ALU. This can be a register (on the R input), an immediate value (on the IMM input), or data from memory or I/O (on the DIN input).

### 2.3.2 Opcode Implementations

#### NOP

The simplest instruction is the NOP instruction which does - nothing. The default values set for all outputs do nothing either so there is no extra VHDL code needed for this instruction.

#### 8-bit Monadic

We call an instruction monadic if its opcode contains one register number and if the instructions reads the register before computing a new value for it.
Only items 1. and 2. in our checklist apply. The default value for DDDDD is already correct. Thus only ALU_OP, WE_D, and WE_F need to be set. We take the DEC Rd instruction as an example:

```
--
--  1001 010d dddd 1010 - DEC
--
Q_ALU_OP <= ALU_DEC;
Q_WE_D <= "01";
Q_WE_F <= '1';
```

### 8-bit Dyadic

We call an instruction dyadic if its opcode contains two data sources (a data source being a register number or an immediate operand). As a consequence of the two data sources, dyadic instructions occupy a larger fraction of the opcode space than monadic functions. We take the ADD Rd, Rr opcode as an example. Compared to the monadic functions now item 3. in the checklist applies as well. This would mean we have to set RRRRR but by chance the default value is already correct. Therefore:

```
--
-- 0000 11rd dddd rrrr - ADD
--
Q_ALU_OP <= ALU_ADD;
Q_WE_D <= "01";
Q_WE_F <= '1';
```

### Multiplication

There is a zoo of multiplication instructions that differ in the signedness of their operands (MUL, MULS, MULSU) and in whether the final result is shifted (FMUL, FMULS, and FMULSU) or not. The opcode decoder sets certain bits in the IMM signal to indicate the type of multiplication:

| IMM(7) | shift |
|---|---|
| IMM(6) | Rd is signed |
| IMM(5) | Rr is signed |

```
--
-- 0000 0011 0ddd 0rrr - _MULSU SU "010"
-- 0000 0011 0ddd 1rrr - FMUL UU "100"
-- 0000 0011 1ddd 0rrr - FMULS SS "111"
-- 0000 0011 1ddd 1rrr - FMULSU SU "110"
--
Q_DDDDD(4 downto 3) <= "10"; -- regs 16 to 23
Q_RRRRR(4 downto 3) <= "10"; -- regs 16 to 23
Q_ALU_OP <= ALU_MULT;
if I_OPC(7) = '0' then
    if I_OPC(3) = '0' then
        Q_IMM(7 downto 5) <= MULT_SU;
    else
        Q_IMM(7 downto 5) <= MULT_FUU;
    end if;
else
    if I_OPC(3) = '0' then
        Q_IMM(7 downto 5) <= MULT_FSS;
    else
        Q_IMM(7 downto 5) <= MULT_FSU;
```

```
    end if;
end if;
Q_WE_01 <= '1';
Q_WE_F <= '1';
```

### Jump and Call

The simplest case of a jump instruction is JMP, an unconditional jump to an absolute address:
The target address of the jump follows after the instruction. Due to our odd/even trick with the program memory, the target address is provided on the upper 16 bits of the opcode and we need not wait for it. We copy the target address from the upper 16 bits of the opcode to the IMM output. Then we set PC_OP to PC_LD_I:

```
--
--  1001 010k kkkk 110k - JMP (k = 0 for 16 bit)
--  kkkk kkkk kkkk kkkk
--
Q_PC_OP <= PC_LD_I;
```

There is a number of conditional jump instructions that differ by the bit in the status register that controls whether the branch is taken or not. BRCS and BRCC branch if bit 0 (the carry flag) is set resp. cleared. BREQ and BRNE branch if bit 1 (the zero flag) is set resp. cleared, and so on.
There is also a generic form where the bit number is an operand of the opcode. BRBS branches if a status register flag is set while BRBC branches if a bit is cleared. This means that BRCS, BREQ, ... are just different names for the BRBS instruction, while BRCC, BRNE, ... are different name for the BRBC instruction.
The relative address (i.e. the offset from the PC) for BRBC/BRBS is shorter (7 bit) than for RJMP (12 bit). Therefore the sign bit of the offset is replicated more often in order to get a 16-bit signed offset that can be added to the PC.

```
--
-- 1111 00kk kkkk kbbb - BRBS
-- 1111 01kk kkkk kbbb - BRBC
--       v
-- bbb: status register bit
-- v: value (set/cleared) of status register bit
--
Q_JADR <= I_PC + (I_OPC(9) & I_OPC(9) & I_OPC(9) & I_OPC(9)
               & I_OPC(9) & I_OPC(9) & I_OPC(9) & I_OPC(9)
               & I_OPC(9) & I_OPC(9 downto 3)) + X"0001";
Q_PC_OP <= PC_BCC;
```

### 2.3.3 List of implemented instructions

## 2.4 Executing

### 2.4.1 Reister File

The processor has 32 general purpose 8-bit registers. Most opcodes use individual 8-bit registers, but some that use a pair of registers. The first register of a register pair is always an even register, while the other register of a pair is the next higher odd register. Instead of using 32 8-bit registers, we use 16 16-bit register pairs. Each register pair consists of two 8-bit registers.

#### General Registers

```
entity reg_16 is
    port ( I_CLK      : in  std_logic;

           I_D        : in  std_logic_vector (15 downto 0);
           I_WE       : in  std_logic_vector ( 1 downto 0);

           Q          : out std_logic_vector (15 downto 0));
end reg_16;
```

The Q output provides the current value of the register pair. There is no need for a read strobe, because (unlike I/O devices) reading the current value of a register pair has no side effects.

#### Status Register

The status register is an 8-bit register. This register can be updated by writing to address 0x5F. Primarily it is updated, however, as a side effect of the execution of ALU operations. If, for example, an arithmetic/logic instruction produces a result of 0, then the zero flag (the second bit in the status register) is set. An arithmetic overflow in an ADD instruction causes the carry bit to be set, and so on. The status register is declared as:

```
entity status_reg is
    port ( I_CLK      : in  std_logic;

           I_COND     : in  std_logic_vector ( 3 downto 0);
           I_DIN      : in  std_logic_vector ( 7 downto 0);
           I_FLAGS    : in  std_logic_vector ( 7 downto 0);
           I_WE_F     : in  std_logic;
           I_WE_SR    : in  std_logic;
```

```
            Q           : out std_logic_vector ( 7 downto 0);
            Q_CC        : out std_logic);
end status_reg;
```

If WE_FLAGS is '1' then the status register is updated as a result of an ALU operation; the new value of the status register is provided on the FLAGS input which comes from the ALU.

If WE_SR is '1' then the status register is updated as a result of an I/O write operation (like OUT or STS); the new value of the status register is provided on the DIN input.

The output Q of the status register holds the current value of the register. In addition there is a CC output that is '1' when the condition indicated by the COND input is fulfilled. This is used for conditional branch instructions. COND comes directly from the opcode for a branch instruction (bit 10 of the opcode for the "polarity" and bits 2-0 of the opcode for the bit of the status register that is being tested).

### 2.4.2 ALU

The first step in the computation of the arithmetic and logic functions is to compute a number of helper values. The reason for computing them beforehand is that we need these values several times, either for different but similar opcodes (e.g. CMP and SUB) but also for the result and for the flags of the same opcode.

```
L_ADIW_D <= I_D + ("0000000000" & I_IMM(5 downto 0));
L_SBIW_D <= I_D - ("0000000000" & I_IMM(5 downto 0));
L_ADD_DR <= L_D8 + L_RI8;
L_ADC_DR <= L_ADD_DR + ("0000000" & I_FLAGS(0));
L_ASR_D  <= L_D8(7) & L_D8(7 downto 1);
L_AND_DR <= L_D8 and L_RI8;
L_DEC_D  <= L_D8 - X"01";
L_INC_D  <= L_D8 + X"01";
L_LSR_D  <= '0' & L_D8(7 downto 1);
L_NEG_D  <= X"00" - L_D8;
L_NOT_D  <= not L_D8;
L_OR_DR  <= L_D8 or L_RI8;
L_PROD   <= (L_SIGN_D & L_D8) * (L_SIGN_R & L_R8);
L_ROR_D  <= I_FLAGS(0) & L_D8(7 downto 1);
L_SUB_DR <= L_D8 - L_RI8;
L_SBC_DR <= L_SUB_DR - ("0000000" & I_FLAGS(0));
L_SIGN_D <= L_D8(7) and I_IMM(6);
L_SIGN_R <= L_R8(7) and I_IMM(5);
L_SWAP_D <= L_D8(3 downto 0) & L_D8(7 downto 4);
L_XOR_DR <= L_D8 xor L_R8;
```

Most values should be obvious, but a few deserve an explanation: There is a considerable number of multiplication functions that only differ in the signedness of their operands. Instead of implementing a different 8-bit multiplier for each opcode, we use a common signed 9-bit multiplier for all opcodes. The opcode decoder sets bits 6 and/or 5 of the IMM input if the D operand and/or the R operand is signed. The signs of the operands

are then SIGN_D and SIGN_R; they are 0 for unsigned operations. Next the signs are prepended to the operands so that each operand is 9-bit signed. If the operand was unsigned (and the sign was 0) then the new signed 9-bit operand is positive. If the operand was signed and positive (and the sign was 0 again) then the new operand is positive again. If the operand was signed and negative, then the sign was 1 and the new operand is also negative.

### Output and Flag Multiplexing

The necessary computations in the ALU have already been made in the previous section. What remains is to select the proper result and setting the flags. The output DOUT and the flags are selected by ALU_OP. We take the first two values of ALU_OP as an example and leave the remaining ones as an exercise for the reader.

```
process(L_ADC_DR, L_ADD_DR, L_ADIW_D, I_ALU_OP, L_AND_DR, L_ASR_D,
       I_BIT, I_D, L_D8, L_DEC_D, I_DIN, I_FLAGS, I_IMM, L_MASK_I,
       L_INC_D, L_LSR_D, L_NEG_D, L_NOT_D, L_OR_DR, I_PC, L_PROD,
       I_R, L_RI8, L_RBIT, L_ROR_D, L_SBIW_D, L_SUB_DR, L_SBC_DR,
       L_SIGN_D, L_SIGN_R, L_SWAP_D, L_XOR_DR)
begin
    Q_FLAGS(9) <= L_RBIT xor not I_BIT(3);  -- DIN[BIT] = BIT[3]
    Q_FLAGS(8) <= ze(L_SUB_DR);             -- D == R for CPSE
    Q_FLAGS(7 downto 0) <= I_FLAGS;
    L_DOUT <= X"0000";

    case I_ALU_OP is
        when ALU_ADC =>
            L_DOUT <= L_ADC_DR & L_ADC_DR;
            Q_FLAGS(0) <= cy(L_D8(7), L_RI8(7), L_ADC_DR(7)); -- Carry
            Q_FLAGS(1) <= ze(L_ADC_DR);                       -- Zero
            Q_FLAGS(2) <= L_ADC_DR(7);                        -- Negative
            Q_FLAGS(3) <= ov(L_D8(7), L_RI8(7), L_ADC_DR(7)); -- Overflow
            Q_FLAGS(4) <= si(L_D8(7), L_RI8(7), L_ADC_DR(7)); -- Signed
            Q_FLAGS(5) <= cy(L_D8(3), L_RI8(3), L_ADC_DR(3)); -- Halfcarry

        when ALU_ADD =>
            L_DOUT <= L_ADD_DR & L_ADD_DR;
            Q_FLAGS(0) <= cy(L_D8(7), L_RI8(7), L_ADD_DR(7)); -- Carry
            Q_FLAGS(1) <= ze(L_ADD_DR);                       -- Zero
            Q_FLAGS(2) <= L_ADD_DR(7);                        -- Negative
            Q_FLAGS(3) <= ov(L_D8(7), L_RI8(7), L_ADD_DR(7)); -- Overflow
            Q_FLAGS(4) <= si(L_D8(7), L_RI8(7), L_ADD_DR(7)); -- Signed
            Q_FLAGS(5) <= cy(L_D8(3), L_RI8(3), L_ADD_DR(3)); -- Halfcarry
```

First of all, the default values for the flags and the ALU output are chosen. The default of L_OUT is 0, while the default for O_FLAGS is I_FLAGS. This means that all flags that are not explicitly changed remain the same. The upper two flag bits are set according to

specific needs of certain skip instructions (CPSE, SBIC, SBIS, SBRC, and SBRS).
Then comes a big case statement for which we explain only the first two cases, ALU_ADC and ALU_ADD.
The expected value of DOUT was already computed as L_ADC_DR in the previous section and this value is assigned to DOUT.
After that the flags that can change in the execution of the ADC opcode are computed. The computation of flags is very similar for a number of different opcodes. We have therefore defined functions cy(), ze(), ov(), and si() for the usual way of computing these flags:
The half-carry flags is computed like the carry flag but on bits 3 rather than bits 7 of the operands and result.
The next example is ADD. It is similar to ADC, but now L_ADD_DR is used instead of L_ADC_DR.

### 2.4.3 List of ALU Operations

| ALU_OP | DOUT | Size |
|---|---|---|
| ALU_ADC | D + R + Carry | 8-bit |
| ALU_ADD | D + R | 8-bit |
| ALU_ADIW | D + IMM | 16-bit |
| ALU_AND | D and R | 8-bit |
| ALU_ASR | D >>1 | 8-bit |
| ALU_BLD | T-flag << IMM | 8-bit |
| ALU_BST | (set T-flag) | 8-bit |
| ALU_COM | not D | 8-bit |
| ALU_DEC | D - 1 | 8-bit |
| ALU_EOR | D xor R | 8-bit |
| ALU_IN | DIN | 8-bit |
| ALU_INC | D + 1 | 8-bit |
| ALU_LSR | D >> 1 | 8-bit |
| ALU_D_MOV_Q | D | 16-bit |
| ALU_R_MOV_Q | R | 16-bit |
| ALU_MULT | D * R | 16-bit |
| ALU_NEG | 0 - D | 8-bit |
| ALU_OR | A or R | 8-bit |
| ALU_PC | PC | 16-bit |
| ALU_PC_1 | PC + 1 | 16-bit |
| ALU_PC_2 | PC + 2 | 16-bit |
| ALU_ROR | D rotated right | 8-bit |
| ALU_SBC | D - R - Carry | 8-bit |
| ALU_SBIW | D - IMM | 16-bit |
| ALU_SREG | (set a flag) | 8-bit |
| ALU_SUB | D - R | 8-bit |
| ALU_SWAP | D[3:0] & D[7:4] | 8-bit |

## 2.5 Design Tools

The tools for our project are briefly described below. But due to the size limitation, not all of these tools' image are presented.

| Code Editing | Version Control | Simulate and Synthesis |
|---|---|---|
| VIM<br>Sigasi | GitHub | GHDL<br>Vivado |

- VIM is one of the best and hardest tools for writing codes. But it needs a lot of customization and have a deep learning curve. So it is generally used by advanced developers and power users. Personally, I choose VIM to edit everything.

- Sigasi is a another choice for VHDL code editing. I have tried this for a short time. It's smarter than VIM as it is a very specific tool for only VHDL but it's not a modal text editor like VIM.

- GitHub is a web-based hosting service for software development projects that use the Git revision control system. I put the project on GitHub, so every member in our group can have access to the source codes and have an opportunity to make changes. The url for our project is: https://github.com/d8660091/ELG5195Project. You can download all the schematics and source codes here.

- GHDL is a very limited but free simulation tool for VHDL codes. It is used at first for our project, but as it's lacks some critical function like generating schematics and very hard to use. It need another application gtkwave to view the simulation result, cause GHDL doesn't have GUI. Vivado is adopted later.

- Vivado is the best choice I found after disappointed by GHDL. It has all the feature I want like RTL analysis, synthesis and implementation. The most important part is that it's very easy to use. It's free if your project is not big. But it only works on Windows and Linux.

| Mnemonics | Operands | Description | Operation | Flags | #Clocks | #Clocks XMEGA |
|---|---|---|---|---|---|---|
| Arithmetic and Logic Instructions | | | | | | |
| ADD | Rd, Rr | Add without Carry | Rd ← Rd+Rr | Z,C,N,V,S,H | 1 | |
| ADC | Rd, Rr | Add with Carry | Rd ← Rd+Rr+C | Z,C,N,V,S,H | 1 | |
| ADIW | Rd, K | Add Immediate to Word | Rd ← Rd + 1:Rd + K | Z,C,N,V,S | 2 | |
| SUB | Rd, Rr | Substract without Carry | Rd ← Rd-Rr | Z,C,N,V,S,H | 1 | |
| SUBI | Rd, K | Substract Immediate | Rd ← Rd-K | Z,C,N,V,S,H | 1 | |
| SBC | Rd, Rr | Substract with Carry | Rd ← Rd-Rr-C | Z,C,N,V,S,H | 1 | |
| SBCI | Rd, K | Substract Immediate with Carry | Rd ← Rd-K-C | Z,C,N,V,S,H | 1 | |
| SBIW | Rd, K | Substract Immediate from Word | Rd+1:Rd ← Rd +1:Rd-K | Z,C,N,V,S | 2 | |
| AND | Rd, Rr | Logical AND | Rd ← Rd·Rr | Z,N,V,S | 1 | |
| ANDI | Rd, K | Logical AND with Immediate | Rd ← Rd·k | Z,N,V,S | 1 | |
| OR | Rd, Rr | Logical OR | Rd ← Rd v Rr | Z,N,V,S | 1 | |
| ORI | Rd, K | Logical OR with Immediate | Rd ← Rd v K | Z,N,V,S | 1 | |
| EOR | Rd, Rr | Exclusive OR | Rd ← Rd⊕Rr | Z,N,V,S | 1 | |
| COM | Rd | One's Complement | Rd ← $FF-Rd | Z,C,N,V,S | 1 | |
| NEG | Rd | Two's Complement | Rd ← $00-Rd | Z,C,N,V,S,H | 1 | |
| INC | Rd | Increment | Rd ← Rd+1 | Z,N,V,S | 1 | |
| DEC | Rd | Decrement | Rd ← Rd-1 | Z,N,V,S | 1 | |
| MUL | Rd, Rr | Multiply Unsigned | R1:R0 ← Rd x Rr (UU) | Z,C | 2 | |
| MULS | Rd, Rr | Multiply Signed | R1:R0 ← Rd x Rr (SS) | Z,C | 2 | |
| MULSU | Rd, Rr | Multiply Signed with Unsigned | R1:R0 ← Rd x Rr (SU) | Z,C | 2 | |
| FMUL | Rd, Rr | Fractional Multiply Unsigned | R1:R0 ← Rd x Rr<<1 (UU) | Z,C | 2 | |
| FMULS | Rd, Rr | Fractional Multiply Signed | R1:R0 ← Rd x Rr<<1 (SS) | Z,C | 2 | |
| FMULSU | Rd, Rr | Fractional Multiply Signed with Unsigned | R1:R0 ← Rd x Rr<<1 (SU) | Z,C | 2 | |
| Branch Instructions | | | | | | |
| RJMP | K | Relative Jump | PC ← PC + k + 1 | None | 2 | |
| IJMP | | Indirect Jump to (Z) | PC(15:0) ← Z, PC(21:16) ← 0 | None | 2 | |

Figure 2.2: Instruction Implemented (1)

| | | | | | | |
|---|---|---|---|---|---|---|
| EIJMP | | Extended Indirect Jump to (Z) | PC(15:0) ← Z, PC(21:16) ← EIND | None | 2 | |
| JMP | k | Jump | PC ← K | None | 3 | |
| RCALL | k | Relative Call Subroutine | PC ← PC + k + 1 | None | 3 / 4(3)(5) | 2 / 3(3) |
| ICALL | | Indirect Call to (Z) | PC(15:0) ← Z, PC(21:16) ← 0 | None | 3 / 4(3) | 2 / 3(3) |
| EICALL | | Extended Indirect Call to (Z) | PC(15:0) ← Z, PC(21:16) ← EIND | None | 4 (3) | 3 (3) |
| CALL | k | call Subroutine | PC ← k | None | 4 / 5(3) | 3 / 4(3) |
| CPSE | Rd, Rr | Compare, Skip if Equal | if (Rd = Rr) PC ← PC + 2 or 3 | None | 1 / 2 / 3 | |
| CP | Rd, Rr | Compare | Rd - Rr | Z,C,N,V,S,H | 1 | |
| CPC | Rd, Rr | Compare with Carry | Rd - Rr - C | Z,C,N,V,S,H | 1 | |
| CPI | Rd, K | Compare with Immediate | Rd - K | Z,C,N,V,S,H | 1 | |
| BRBS | s, k | Branch if Status Flag Set | if (SREG(s) = 1) then PC ← PC + k + 1 | None | 1/2 | |
| BRBC | s, k | Branch if Status Flag Cleared | if (SREG(s) = 0) then PC ← PC + k + 1 | None | 1/2 | |
| Data Transfer Instructions | | | | | | |
| MOV | Rd, Rr | Copy Register | Rd ← Rr | None | | |
| MOVW | Rd, Rr | Copy Register Pair | Rd+1:Rd ← Rr +1:Rr | None | | |
| LDI | Rd, K | Load Immediate | Rd ← K | None | 1(5)/2(3) | 2(3)(4) |
| LDD | Rd, Y+q | Load Indirect with Displacement | Rd ← (Y + q) | None | 2(3) | 2(3)(4) |
| STD | Y+q, Rr | Store Indirect with Displacement | (Y + q) ← Rr | None | 2(3) | 2(3) |
| Bit and Bit-test Instructions | | | | | | |
| LSR | Rd | Logical Shift Left | Rd(n) ← Rd(n +1), Rd(7) ← 0, C ← Rd(7) | Z,C,N,V | 1 | |
| ROR | Rd | Rotate Left Through Carry | Rd(7) ← C, Rd(n) ← Rd(n +1), C ← Rd(0) | Z,C,N,V | 1 | |
| ASR | Rd | Arithmetic Shift Right | Rd(n) ← Rd(n +1), n=0..6 | Z,C,N,V | 1 | |
| SWAP | Rd | Swap Nibbles | Rd(3..0) ↔ Rd(7..4) | None | 1 | |

Figure 2.3: Instruction Implemented (2)

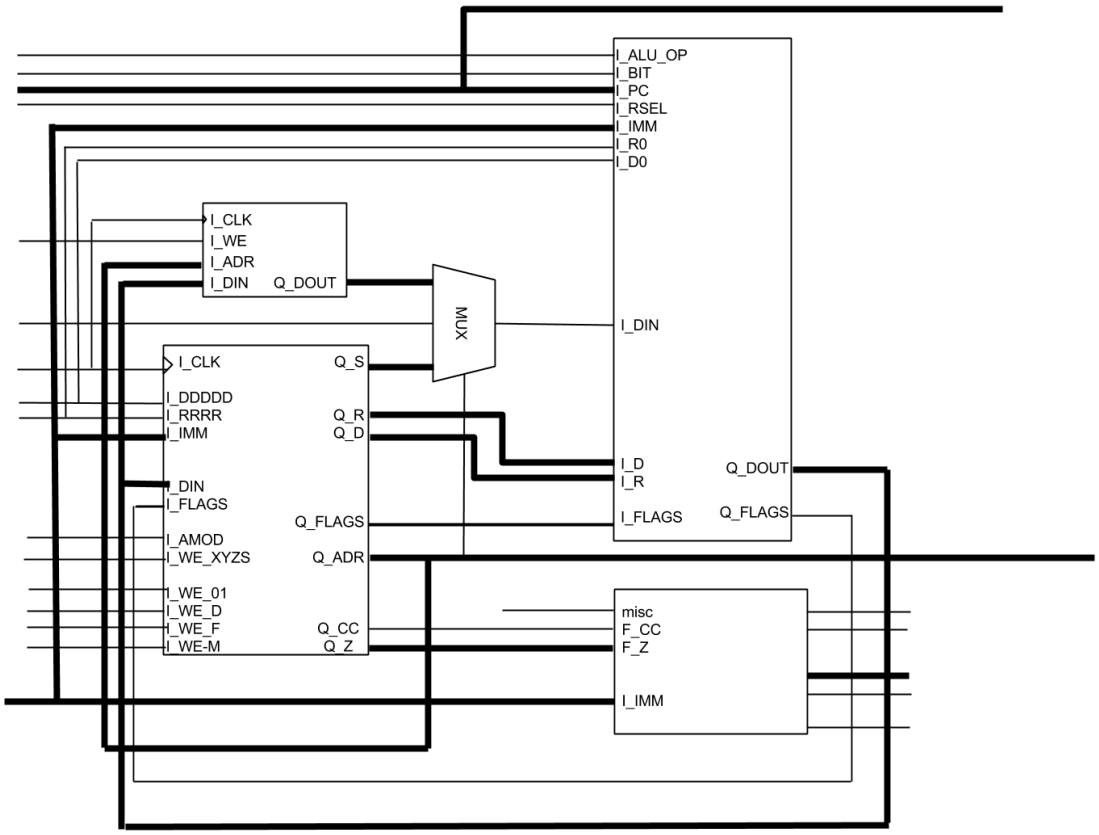



Figure 2.4: Datapath schematics

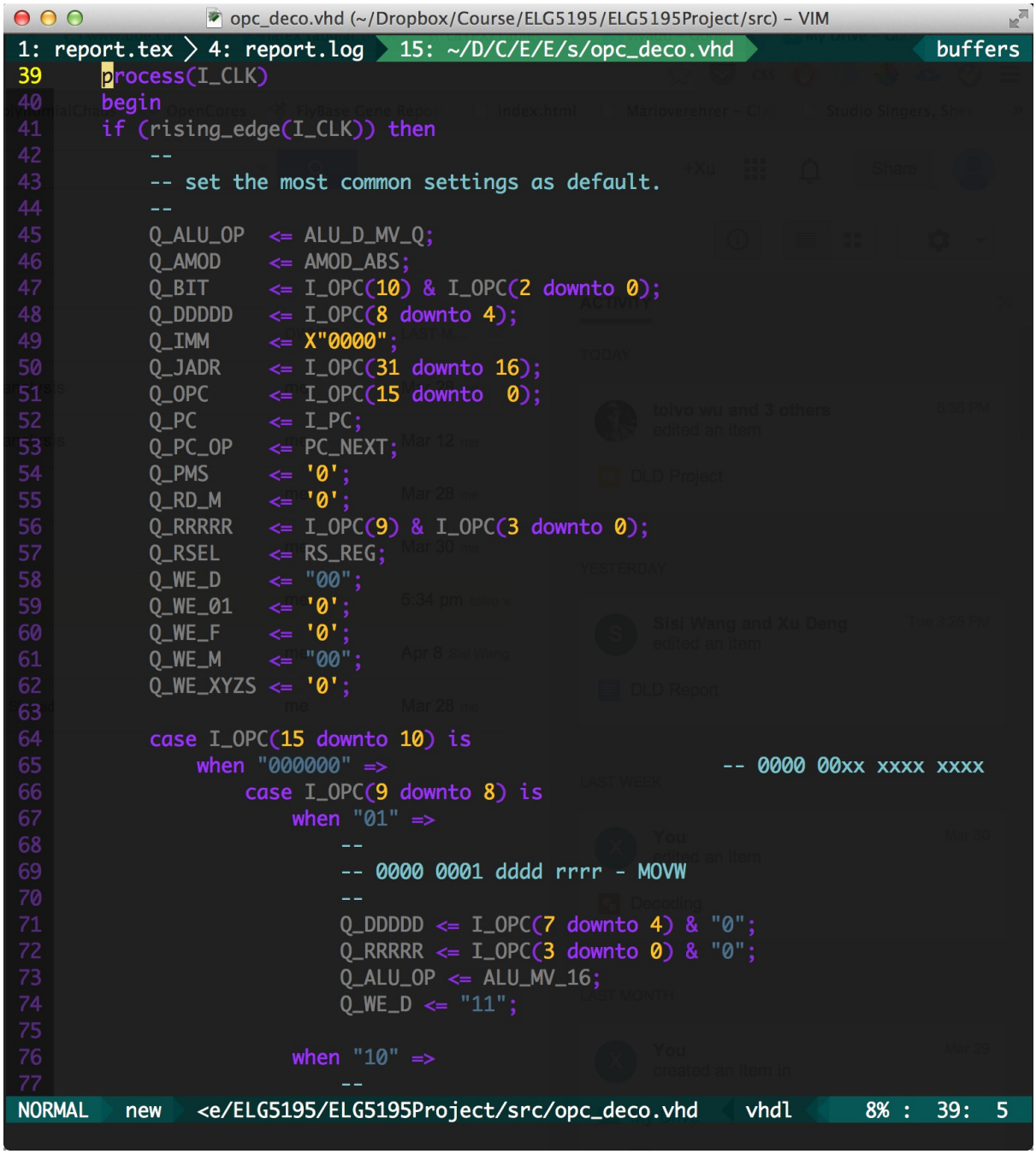



Figure 2.5: VIM

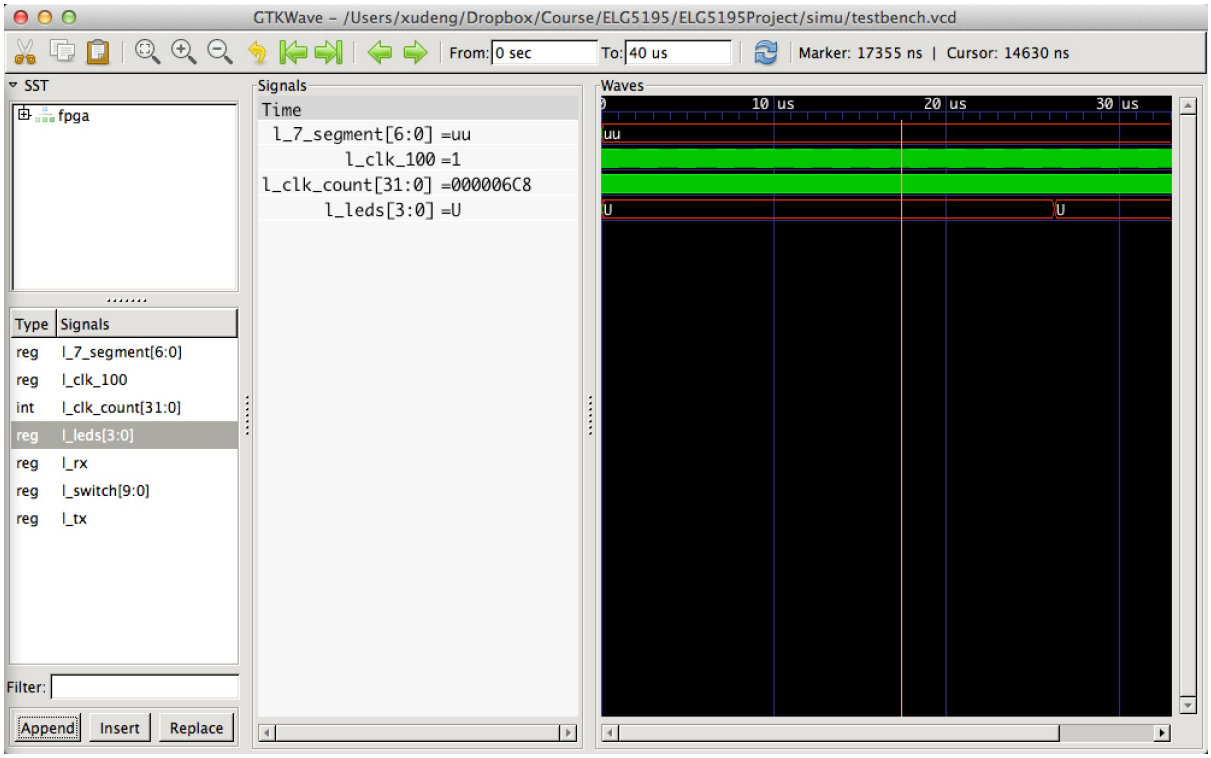
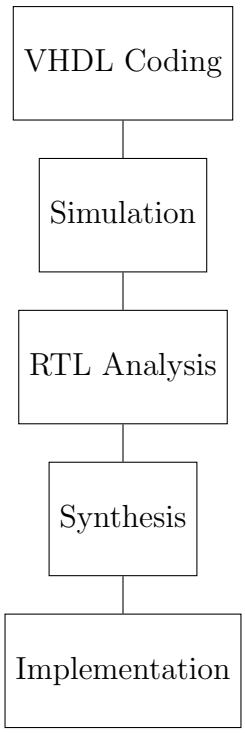



Figure 2.6: gtkwave for viewing .vcd files generated by GHDL

# Chapter 3

# Results Dissemination

## 3.1 Validation Procedure




## 3.2 Simulation/Synthesis Results

A testbench is used here to do the behavioural simulation of the pipelined processor. In this file, the inputs of CPU clock and clear are connected to local signals L_CLK and L_CLR respectively. The signal L_CLK changes itself between 0 1 every 5 ns. The signal L_CLR equals to 1 in the first 2 clocks in order to initiate the processor, then always equals 0 for the future clocks. We should notice here, that this testbench didn't include the peripheral modules such as I/O or segment display. This is rational because in our project, we only concern about the logic in the CPU, not to implement that in real circuit.

```
library IEEE;
```

```
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity testbench is
end testbench;

architecture Behavioral of testbench is

component cpu_core
    port ( I_CLK      : in  std_logic;
           I_CLR      : in  std_logic;
           I_INTVEC   : in  std_logic_vector( 5 downto 0);
           I_DIN      : in  std_logic_vector( 7 downto 0);

           Q_OPC      : out std_logic_vector(15 downto 0);
           Q_PC       : out std_logic_vector(15 downto 0);
           Q_DOUT     : out std_logic_vector( 7 downto 0);
           Q_ADR_IO   : out std_logic_vector( 7 downto 0);
           Q_RD_IO    : out std_logic;
           Q_WE_IO    : out std_logic);

end component;

signal L_CLK          : std_logic;
signal L_CLR          : std_logic;
signal L_CLK_COUNT       : integer := 0;

begin

    cpu: cpu_core
    port map(
              I_CLK  => L_CLK,
              I_CLR  => L_CLR,
              I_INTVEC => "000000",
              I_DIN => X"00",

              Q_OPC => open,
              Q_PC  => open,
              Q_DOUT => open,
              Q_ADR_IO => open,
              Q_RD_IO => open,
              Q_WE_IO => open
            );
    process
    begin
        clock_loop : loop
            L_CLK <= transport '0';
```

```
            wait for 5 ns;

            L_CLK <= transport '1';
            wait for 5 ns;
        end loop clock_loop;
    end process;

    process(L_CLK)
    begin
        if (rising_edge(L_CLK)) then
          if(L_CLK_COUNT < 2) then
            L_CLR <= '1';
          else
            L_CLR <= '0';
          end if;
          L_CLK_COUNT <= L_CLK_COUNT + 1;
        end if;
    end process;
end Behavioral;
```

The program memory used for testing is presented below. It consists of only 5 instructions, but it doesn't mean that the processor can only work for theses 5 instructions. In fact, due to the time limitation of this project, it's impossible to write a program which can test all the instruction in one time. But this can be the work for the future.
Another thinking of testing the processor is to load it with real program, which means test it with the instructions generated by the compiler. However, there are several limitations. First, we didn't implement the peripheral circuit, which is quite ordinary in general programs. Second, implement all the instruction sets takes a long time and effort. Third, we need to find a way to load the binary file which is generated by the compiler to the VHDL file.
These 5 codes used here because it's easy to understand and can show the proper working of our processor. Other instructions are tested separately and the result proved our design.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;


package prog_mem_content is
constant p_00 : std_logic_vector := X"E080"; -- LOAD: R24 <- 0
constant p_01 : std_logic_vector := X"9601"; -- ADIW: R24 <- R24+1
constant p_02 : std_logic_vector := X"2FA8"; -- MOV: R26 <- R24
constant p_03 : std_logic_vector := X"50AA"; -- SUBI: R26 <- 26-10
constant p_04 : std_logic_vector := X"940C"; -- JMP: to start 32 bits
constant p_05 : std_logic_vector := X"0001";
constant p_06 : std_logic_vector := X"FFFF";
constant p_07 : std_logic_vector := X"FFFF";
constant p_08 : std_logic_vector := X"FFFF";
```

```
constant p_09 : std_logic_vector := X"FFFF";
constant p_0A : std_logic_vector := X"FFFF";
end prog_mem_content;
```

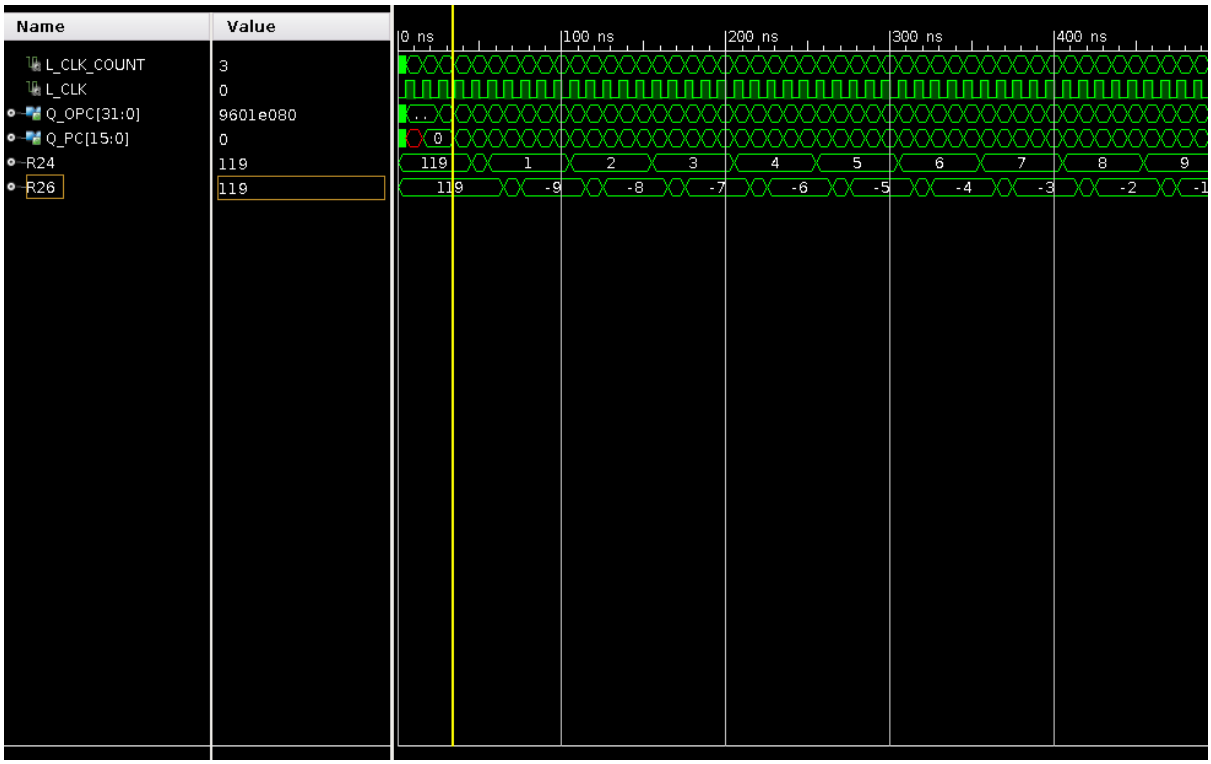



Figure 3.1: Simulation Results

## 3.3 Discussion

Like the waveform showed in 3.1. The general register R24 and R26 is used as a counter. Their initial value is 119. Then both register increase by 1 every 5 clocks. We can see that for the R26, there is one clock in which the value of R26 is different than the value of R24-10. This is caused by the fact that we used two instructions to do this. We first move the R24 to R26, then minus R26-10. So there is one clock in which the value of R26 is different.
Though the logic of our processor design is proved. I didn't find a way to do the timing test through a lot of effort. This may be caused by some architecture we use to implement entities. Future work should be done to resolve this problem.

## 3.4 Conclusion

In this project, we designed a basic pipelined CPU. The word "basic" is used here because we only designed the CPU core, not including peripheral circuit, so it can't be applied in practical. However, once the CPU core has been designed, the peripheral circuit is very easy to add.
The main features of our pipelined CPU including:

1. Using dual port memory to solve the structural hazards.
2. The architecture make it possible that register file and data memory is located in the same stage (executing stage). Thus data hazards are avoided.
3. A SKIP signal connect the executing stage and opcode fetching stage, which affects the opcode output immediately.
4. Quite large set of actual instruction is implemented.

Future work including:

1. Implement more instructions.
2. Test with real programs generated by compiler.
3. Design peripheral circuit.