

COMP 3211 Final Report

z5210146 Chung Lai Lee
z5270589 YiJie Zhao

The detail (type/register/cycle) of each of your instructions, and how to use the instruction (you can provide an example code of that instruction):

Our pattern search processor uses a more software approach. Our processor uses a 32 bit size instruction set, where the opcode and operand are 8 bits each. The machine has 32 16-bit registers, where register \$0 - \$15 are specialized register used to store the variables of the program, \$16 - \$31 stores the input string that we want to search pattern, due to time limitation, some of our register will be hard coded, such as the length of the input string, which is stored in \$10. The address space for instruction is 8 bits so it allows a maximum of 256 instructions.

Data memory component which is located in the MEM stage has 3 subcomponents - pattern character array (PA), pattern length array (LA) and pattern occurrence array (OA). All of them are 16 bits addressable.

Below is the ISA of our processor:

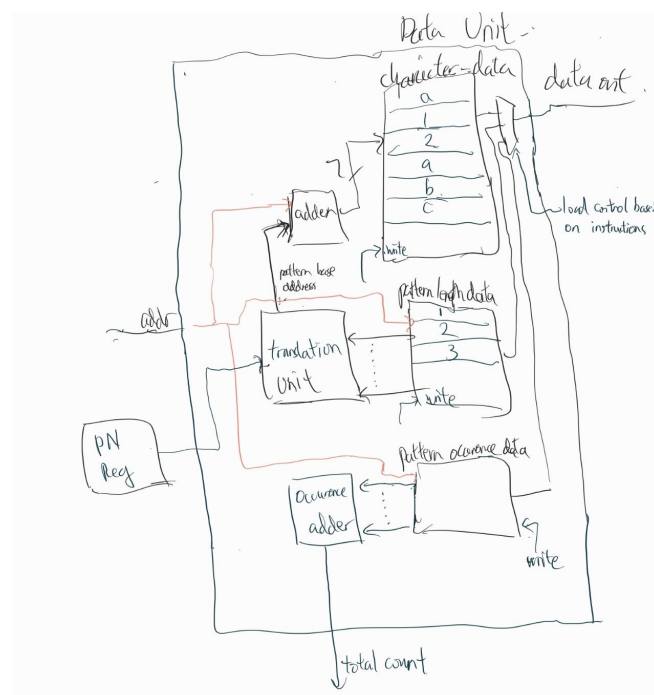
BNE	R1 R2 imm	if (R1 != R2) branch to PC + 1 + imm
ADD	R1 R2 R3	R3 <- R1 + R2
SOW	R1 R2 imm	OA[R1+imm] <- R2
LOW	R1 R2 imm	R2 <- OA[R1+imm]
STB	R1 X X	PNReg <- R1, PNreg is a specialized register in MEM stage that controls offset used in instruction LPA.
LPA	R1 R2 R3	if (R2 < R1) {R3 <- PA[offset + R2] and R2++} else {R3 <- -1}, offset here is determined by PNReg
LLA	R1 R2 R3	if (R2 < R1) {R3 <- LA[R2] and R2++} else {R3 <- -1}
LOA	R1 R2 R3	if (R2 < R1) {R3 <- OA[R2] and R2++} else {R3 <- -1}
MVI	R1 X R2	R2 <- reg_file[R1]
END	X X X	signal end of function (stalls pipeline and signals ready when END instruction reaches mem stage)
JP	X X imm	branch to PC + 1 + imm

Since we have different data subcomponents, there are different instructions to load and store data.

LOW and SOR load from and stores to OA respectively.

LPA, LLA and LOA load data from OA, LA and PA respectively, but in addition to that, it will also increment R2, which is the index of the data. This allows us to both retrieve the data and increment the index in one instruction.

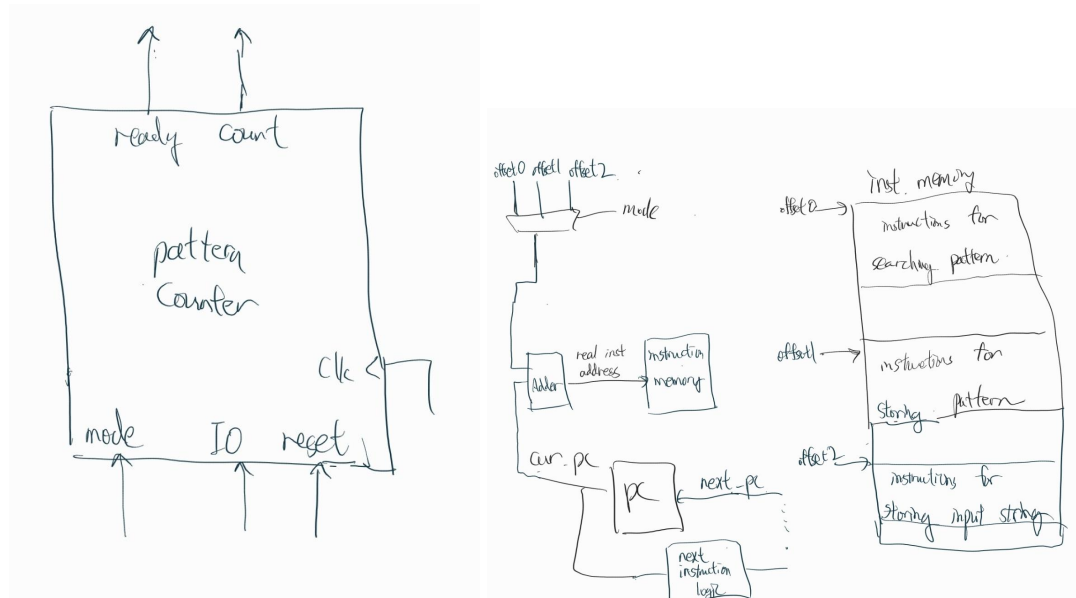
The load of LPA is different from LLA and LOA. In LPA, the R2 register doesn't fully specify the location of the data in the array we want to load. Refer to the ISA above, for instruction LPA, location of the data is offset + R2 instead of simply R2. Offset is determined by the number of PNreg, which is set using instruction STB.



To give an example, suppose the data memory contains the data according to the diagram above (character data in the diagram above is pattern character array PA). We can see that pattern number 0 (P0) is "a", pattern number one (P1) is "1 2" and pattern number two (P2) is "a b c" based on data in the pattern length array.

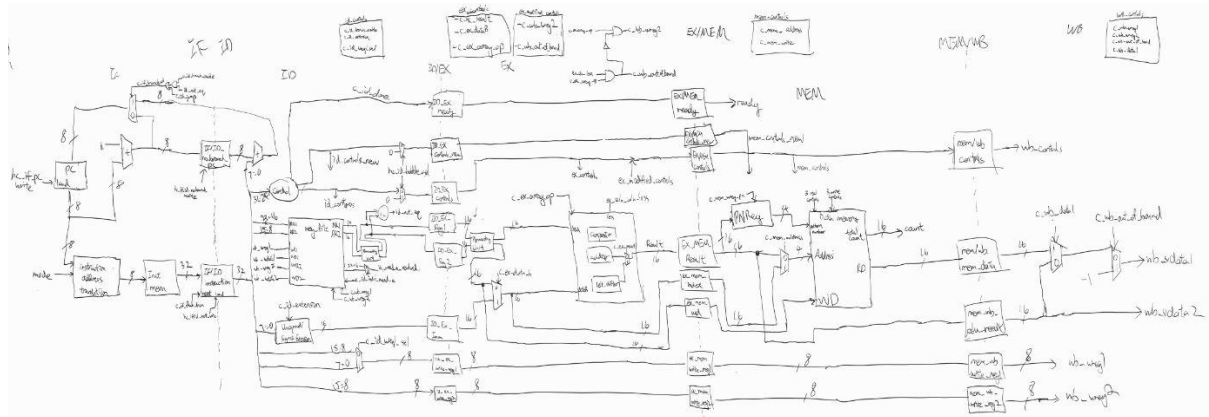
Since all the characters of the patterns are stored in PA, to get the pattern_number_2[1], we need to set PNreg to 2 to indicate that we will be loading pattern number 2's character. Connected to the PNreg is the translation unit, which gets the input data from LA. Since PA is 2, the translation unit adds the first two lengths $1+2 = 3$. This is the value of the offset in the ISA of LPA. Then, if we use LPA with $R2 = 1$, the value of R2 will come through the addr port, which will be added to 3 from the translation unit. The data output will be $PA[4] = b$, which is exactly pattern_number_2[1].

The occurrence count output from OA shown in the diagram above connects to the occurrence adder, which is simply a combination of adders which adds all pattern occurrences together. It outputs total_count, which is the sum of the occurrences of patterns.



We originally planned to split the assignment into a function for storing input string, function for storing patterns and function for searching pattern (when both the pattern and input string is already stored). Although we didn't write the function for storing the input string and pattern, our design allows multiple functions being stored in instruction memory by incorporating an instruction address translation unit. Mode allows different instruction offset for the different functions. The middle diagram shows the connection of the instruction address translation unit and on the right shows the instruction memory layout. In our case, the search function is mode 0, which has an instruction address offset of 0.

An overview of the architecture of your processor:



The diagram above contains all control signals. The naming convention of these signals is as follows:

- Controls
- normal controls: c_[stage]_[signal name]
- hazard controls: hc_[stage]_[signal name]

The [stage] in these signal names specify which stage these signals exert their control on.

For example, c_mem_write_char is the control that controls the write back to the character array (PA) since PA is located at mem stage.

Input and outputs of the forwarding unit and hazard detection unit are not listed in the diagram for the sake of clarity.

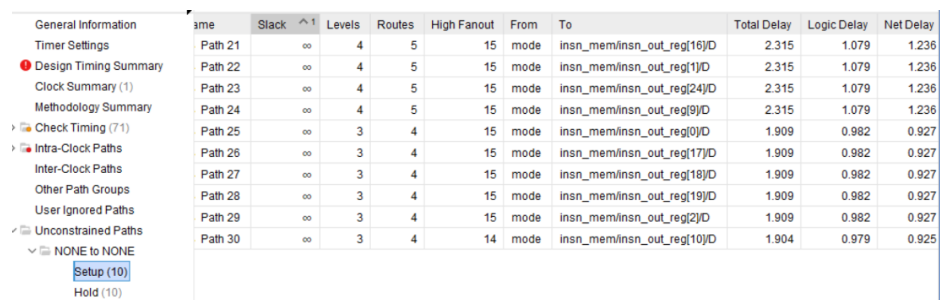
The format of control bus, and the list of controls that is asserted for the different instructions is detailed extensively in control_unit.vhd.

Performance of your processor (frequency, hardware footprint):

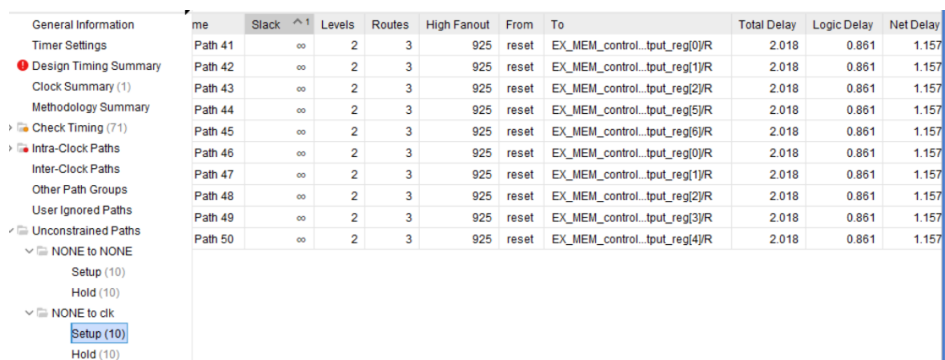
Although the maximum frequency is quite good at 82.8MHz, it doesn't reflect the performance of the pattern search since our searching is a software approach, it takes a lot of clock cycles to finish the task based on pattern length, input string, and whether they match or not.

In terms of hardware, the utilization of LUT, FF and IO are 1.78, 0.74 and 21.18 respectively.

During the time when we were writing out the instruction program for the search function, we released that by using the fact that LLA returns -1 if $RS \geq RT$, it is possible to write an instruction program which doesn't require the translation unit in data memory and PNReg. However, the hardware components were already written and we ran out of time so we didn't change the instruction program. Since the translation unit is primarily composed of adders and mux, that means LUT could be reduced. And in the case of PNReg it's FF.



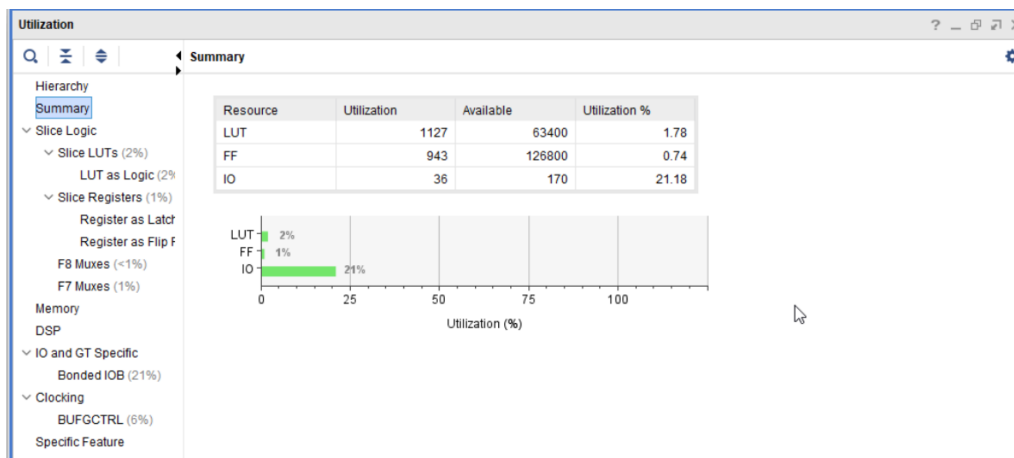
Path Name	Slack	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay
Path 21	∞	4	5	15	mode	insn_mem/insn_out_reg[16]D	2.315	1.079	1.236
Path 22	∞	4	5	15	mode	insn_mem/insn_out_reg[1]D	2.315	1.079	1.236
Path 23	∞	4	5	15	mode	insn_mem/insn_out_reg[24]D	2.315	1.079	1.236
Path 24	∞	4	5	15	mode	insn_mem/insn_out_reg[9]D	2.315	1.079	1.236
Path 25	1.909	3	4	15	mode	insn_mem/insn_out_reg[0]D	1.909	0.982	0.927
Path 26	1.909	3	4	15	mode	insn_mem/insn_out_reg[17]D	1.909	0.982	0.927
Path 27	1.909	3	4	15	mode	insn_mem/insn_out_reg[18]D	1.909	0.982	0.927
Path 28	1.909	3	4	15	mode	insn_mem/insn_out_reg[19]D	1.909	0.982	0.927
Path 29	1.909	3	4	15	mode	insn_mem/insn_out_reg[2]D	1.909	0.982	0.927
Path 30	1.904	3	4	14	mode	insn_mem/insn_out_reg[10]D	1.904	0.979	0.925



Path Name	Slack	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay
Path 41	2.018	2	3	925	reset	EX_MEM_control_tput_reg[0]R	2.018	0.861	1.157
Path 42	2.018	2	3	925	reset	EX_MEM_control_tput_reg[1]R	2.018	0.861	1.157
Path 43	2.018	2	3	925	reset	EX_MEM_control_tput_reg[2]R	2.018	0.861	1.157
Path 44	2.018	2	3	925	reset	EX_MEM_control_tput_reg[5]R	2.018	0.861	1.157
Path 45	2.018	2	3	925	reset	EX_MEM_control_tput_reg[6]R	2.018	0.861	1.157
Path 46	2.018	2	3	925	reset	EX_MEM_control_tput_reg[0]R	2.018	0.861	1.157
Path 47	2.018	2	3	925	reset	EX_MEM_control_tput_reg[1]R	2.018	0.861	1.157
Path 48	2.018	2	3	925	reset	EX_MEM_control_tput_reg[2]R	2.018	0.861	1.157
Path 49	2.018	2	3	925	reset	EX_MEM_control_tput_reg[3]R	2.018	0.861	1.157
Path 50	2.018	2	3	925	reset	EX_MEM_control_tput_reg[4]R	2.018	0.861	1.157

General Information	Name	Slack	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay
Timer Settings	Path 61	∞	10	5	3	data_mem/occur...m_reg[0][1]C	count[15]	7.005	5.277	1.728
Design Timing Summary	Path 62	∞	10	5	3	data_mem/occur...m_reg[0][1]C	count[13]	6.980	5.252	1.728
Clock Summary (1)	Path 63	∞	10	5	3	data_mem/occur...m_reg[0][1]C	count[14]	6.944	5.216	1.728
Methodology Summary	Path 64	∞	10	5	3	data_mem/occur...m_reg[0][1]C	count[12]	6.912	5.184	1.728
Check Timing (71)	Path 65	∞	9	5	3	data_mem/occur...m_reg[0][1]C	count[11]	6.834	5.106	1.728
Intra-Clock Paths	Path 66	∞	9	5	3	data_mem/occur...m_reg[0][1]C	count[10]	6.781	5.053	1.728
Inter-Clock Paths	Path 67	∞	9	5	3	data_mem/occur...m_reg[0][0]C	count[9]	6.647	4.919	1.728
Other Path Groups	Path 68	∞	9	5	3	data_mem/occur...m_reg[0][0]C	count[8]	6.579	4.851	1.728
User Ignored Paths	Path 69	∞	8	5	3	data_mem/occur...m_reg[0][0]C	count[7]	6.501	4.773	1.728
Unconstrained Paths	Path 70	∞	8	5	3	data_mem/occur...m_reg[0][0]C	count[6]	6.448	4.720	1.728
NONE to NONE	Setup (10)									
	Hold (10)									
NONE to clk	Setup (10)									
	Hold (10)									
clk to NONE	Setup (10)									
	Hold (10)									

Utilization:



Setup

Worst Negative Slack (WNS): **-1.077 ns**

Total Negative Slack (TNS): **-20.643 ns**

Number of Failing Endpoints: **26**

Total Number of Endpoints: **1683**

Name	Waveform	Period (ns)	Frequency (MHz)
clk	{0.000 6.000}	12.000	83.333

$$12 - -1.077 = 12.077\text{ns}$$

$$\text{Max frequency} = 1/(12.077\text{ns}) = 82.8\text{MHz}$$

register_file:

-----register hardcoded -----

We use the register 0 to represent the value 0: $\$0 = 0$;

We use the register 1 to represent the value 1: $\$1 = 1$;

We use the register 2 to represent the value -1: $\$2 = -1$;

We use the register 9 to store the total number of patterns:

total_number_of_patterns = $\$9$;

We use the register 10 to represent the length of the input string:

input_string_length = $\$10$;

We use the register 11 to store the base: input_register_base = $\$11 = 16$;

Input_register_base is 16 since $\$16$ stores the first input string character.

This allow us to use

$\$A \leftarrow \$11 + x$

We can then use MVI $\$A \ \13 to move the xth input string character into $\$13$

wildcard = $\$14 = ?$. The character encoding is 0xfe.

We use register 3 to record the number of processed characters:

chars_processed = $\$3 = i$;

We use register 4 to represent the character of current pattern: p_char = $\$4$;

We use the register 5 to record the length of current pattern: curr_pattern_length = $\$5$;

We use the register 6 to record the index of which pattern: curr_pattern_num = $\$6$;

We use the register 7 to store the number of current pattern offset:

curr_pattern_offset = $\$7$;

We use the register 8 to store the number of current input base: curr_input_base = $\$8 (\$8 \leftarrow \$11 + \$3)$;

We use the register 12 to store the index in the current input: curr_input_index = \$12 (\$12 <- \$8 + \$7);

We use register 13 to represent the character of the current input: i_char = \$13 ;

We use the register 15 to record the number of the current pattern occurrence: curr_occurrence = \$15;

The instruction below shows the searching function. (with the input string already pre-loaded in \$16 - \$31, total number of patterns in \$9 and length of input string in \$10. PA and LA are pre-loaded with the stored pattern)

```
var_insn_mem := (others => X"00000000");
-- label 1
var_insn_mem(0) := X"08000006"; -- ADD $0 $0 $6 curr_pattern_num <- 0
-- label 2
var_insn_mem(1) := X"04060000"; -- STB $6 0 0 special_p_register <- curr_pattern_num
var_insn_mem(2) := X"08000007"; -- ADD $0 $0 $7 curr_pattern_offset <- 0
var_insn_mem(3) := X"080b0308"; -- ADD $11 $3 $8 curr_input_base <- input_register_base + chars_processed
var_insn_mem(4) := X"07090605"; -- LLA $9 $6 $5 curr_pattern_length <- LA[curr_pattern_num], curr_pattern_num++
-- label 3
var_insn_mem(5) := X"0808070c"; -- ADD $8 $7 $12 curr_input_index <- curr_input_base + curr_pattern_offset
var_insn_mem(6) := X"00000000";
var_insn_mem(7) := X"00000000";
var_insn_mem(8) := X"00000000";
var_insn_mem(9) := X"0b0e000d"; -- MVI $12 0 $13 i_char <- REG[curr_input_index]
var_insn_mem(10) := X"05050704"; -- LPA $5 $7 $4 p_char <- PA[p_offset+curr_pattern_offset], curr_pattern_offset++
var_insn_mem(11) := X"060d040a"; -- BNE $13 $4 label 6 if (i_char != p_char) goto label 6
-- label 4
var_insn_mem(12) := X"060705f8"; -- BNE $7 $5 label 3 if (curr_pattern_offset != curr_pattern_length) goto label 3:
var_insn_mem(13) := X"0f060fff"; -- LOW $6 $15 -1 curr_occurrence <- OA[--curr_pattern_num]
var_insn_mem(14) := X"080f010f"; -- ADD $15 $1 $15 curr_occurrence++
var_insn_mem(15) := X"02060fff"; -- SOW $6 $15 -1 OA[--curr_pattern_num] <- curr_occurrence
-- label 5
var_insn_mem(16) := X"060609f0"; -- BNE $6 $9 label 2 if (curr_pattern_num != total_number_of_patterns) goto label 2
var_insn_mem(17) := X"08030103"; -- ADD $3 $1 $3 char_process++
var_insn_mem(18) := X"06030aed"; -- BNE $3 $10 label 1 if (chars_processed != input_string_length) goto label 1
var_insn_mem(19) := X"0d000000"; -- END 0 0 0
var_insn_mem(20) := X"00000000"; -- nop
var_insn_mem(21) := X"00000000"; -- nop
-- label 6
var_insn_mem(22) := X"060e0df9"; -- BNE $14 $13 label 5 if (? != i_char) goto label 5
var_insn_mem(23) := X"0e0000f4"; -- JP 0 0 laebl 4 jump to label 4
```

The comments in the instruction memory combined with the registers use detailed how each instruction worked. Although there are a couple points I would like to make:

In instruction 13 and 15, the immediate field of LOW and SOW is -1. This is because previously in instruction 4, LLA automatically increments curr_pattern_num. But since we would like to store back the occurrence count of the same pattern, we set the imm field as -1 to index correctly in OA.

In the MVI instruction in instruction 9, the reason why instruction 6, 7 and 8 are nop is because we didn't get forwarding and stalling to work with MVI. This is because the data source of MVI depends on both R1 and R2. When R2 is

changed, the source of where the data is read is changed as well. This results in the logic of forwarding and hazard detection to become a lot more complex. Due to time constraints, we simply insert bubbles to avoid the issue.