

CSED311: Lab 3 (due Apr. 16)

김태연(20220140), 손량(20220323)

Last compiled on: Tuesday 16th April, 2024, 13:46

1 Introduction

RISC-V architecture를 바탕으로 구성된 Multicycle CPU를 구현한다.

2 Design

이번에 구현한 RISC-V Multicycle CPU는 다음과 같은 주요 기능을 가진 Submodule로 나누었으며, 생성된 값들을 선택하기 위해 추가적으로 Multiplexer를 구현하여 사용했다. 기존 Single-cycle CPU에서 추가로 제작한 Adder는 ALU에서 재사용되었기 때문에 더 이상 필요로 하지 않는다.

2.1 Program Counter

기존 Single-cycle CPU에서는 Program Counter가 Clock signal에 Synchronous하게 무조건적으로 반영되었다. Multicycle CPU의 구현에서는 Clock signal에 Synchronous하면서도, 추가적인 Control signal인 `pc_write` 및 `pc_commit` Signal에 따라 동작한다. `pc_write` Signal은 Jump instruction 혹은 Branch instruction 여부 및 ALU의 Branch condition에 따라 활성화되며, `pc_commit`는 Multicycle CPU의 Instruction 별 Stage에서 Program counter가 갱신되어야 할 시점에 활성화되도록 Control unit에서 구현하였다.

2.2 Memory

기존 Single-cycle CPU에서는 Instruction Memory와 Data Memory가 별도로 분리되어 있었지만, Multicycle CPU에서는 해당 구분 없이 하나의 통합된 $32\text{bit} \times 16384 = 64\text{KiB}$ 의 메모리를 사용한다. Memory 접근 이후 값을 전달할 때, Memory Data Register (`inst_reg`) 및 Instruction Register(`mem_data_reg`)에 전달하게 된다.

2.2.1 Memory Data Register

Skeleton Code에서 주어진 MDR에 Clock signal과 Synchronous하게 Memory 모듈의 `dout`으로부터 나온 데이터를 저장한다. Clock signal에 따라 항상 값을 쓸 수 있다. Memory Data Register의 값은 다음과 같은 용도로 사용된다.

- Register File에 값을 Write back 하기 위한 값을 제공한다.

2.2.2 Instruction Register

Skeleton Code에서 주어진 IR에 Clock signal과 Synchronous하게 Memory 모듈의 dout으로부터 나온 데이터를 저장한다. Memory Data Register와 구현이 유사하나, Instruction Register는 Clock에 Synchronous하면서 Control Unit의 `ctrl_unit_inst_reg_write` Signal이 Enable 되었을 때만 작성될 수 있다. Instruction Register의 값은 다음과 같은 용도로 사용된다.

- Register File에 접근하기 위한 `rs1`, `rs2`, `rd` 와 같은 Register 정보를 제공한다.
- Immediate Generator에서 상수를 생성하는데 사용된다.
- Control Unit 및 ALU Control Unit에서 Control signal을 생성하기 위한 정보를 제공한다.
- Ecall Unit에서 `ecall`을 판단하기 위해 사용된다.

2.3 Register File

Single-cycle CPU와 마찬가지로 Clock signal에 Synchronous하게 Register value를 입력받고 갱신한다.

2.3.1 Operand Register – A, B

Register file의 출력을 Clock signal에 맞추어 저장하는 temporaries이며, 입력된 다음 Cycle에 ALU의 입력으로 (각각의 Multiplexer에서 선택되었을 경우) 그 값을 전달한다.

2.4 ALU

Single-cycle CPU와 마찬가지로 `alu_op`에 따라 연산을 결정하고, Clock signal과 Asynchronous하게 Register value 혹은 Immediate value로부터 값을 계산하여 출력한다. Branch condition의 경우는 Lab2에서 구현한 방식과 동일하게 LSB 1bit에 boolean 연산의 결과를 전달하도록 설계하였다.

2.4.1 ALU Register – ALUOut

ALU의 출력을 Clock signal에 맞추어 ALUOut에 전달하며, (현재 Cycle의 ALU 연산 결과)와 (이전 Cycle의 ALU 연산 결과를 ALUOut에 저장한 값)중 적절한 값을 Register file에 Write back 하거나 Program counter를 갱신하는데 사용한다.

2.5 Immediate Generator

RV32I instruction 중 immediate value를 가지는 instruction의 타입이 존재하며, 각 타입 별 Immediate value가 생성되는 부분들 또한 다르다. Single-cycle CPU와 마찬가지로, Clock signal과 Asynchronous하게 값을 계산하여 출력한다. 각 Instruction opcode의 type에 따라 생성해야 할 Immediate value를 immediate generator의 출력으로 전달한다.

2.6 Control Unit

어떤 Instruction type이던, LSB 7bit는 RV32I에서 opcode로 정의가 되어 있는 부분이다. Clock Signal에 Synchronous하게 Control Unit에서 현재 Instruction과 현재 State로부터 다음 Multicycle CPU Stage로 전환하였다. 이와 병렬적으로, `always @(*)`를 사용하여 각 CPU State 및 Instruction의 조합에서 CPU Control signal을 CPU Clock과 Asynchronous하게 생성하도록 설계하였다.

2.7 Ecall Unit

프로그램의 종료를 판별하기 위한 Ecall의 구조를 정의하였으며, Clock signal과 asynchronous하게 수행된다.

3 State Design

3.1 State Design – Stage

Multicycle CPU를 구현하면서, Instruction이 각 Stage에 다음과 같은 작업을 수행하도록 분할하였다.

3.1.1 R-type Register Arithmetic

State	Action
IF	Instruction Fetch $PC \leftarrow PC + 4$
ID	Instruction Decode $alu_reg \leftarrow PC + imm$
EX	$alu_reg \leftarrow reg_file[rs1] \text{ op } reg_file[rs2]$ (for value)
MEM	<i>Pass</i>
WB	Save alu_reg to $reg_file[rd]$

3.1.2 I-type Immediate Arithmetic

State	Action
IF	Instruction Fetch $PC \leftarrow PC + 4$
ID	Instruction Decode $alu_reg \leftarrow PC + imm$
EX	$alu_reg \leftarrow reg_file[rs1] \text{ op } imm$ (for value)
MEM	<i>Pass</i>
WB	Save alu_reg to $reg_file[rd]$

3.1.3 I-type Load

State	Action
IF	Instruction Fetch $PC \leftarrow PC + 4$
ID	Instruction Decode $alu_reg \leftarrow PC + imm$
EX	$alu_reg \leftarrow reg_file[rs1] \text{ op } imm$ (for address)
MEM	Save memory to mem_data_reg
WB	Save mem_data_reg to $reg_file[rd]$

3.1.4 S-type Store

State	Action
IF	Instruction Fetch $PC \leftarrow PC + 4$
ID	Instruction Decode $alu_reg \leftarrow PC + imm$
EX	$alu_reg \leftarrow reg[rs1] \text{ op } imm$ (for address)
MEM	Save $memory[reg[rs2]] \leftarrow alu_reg$
WB	Pass

3.1.5 B-type Branch

State	Action
IF	Instruction Fetch $PC \leftarrow PC + 4$
ID	Instruction Decode $alu_reg \leftarrow PC + imm$
EX	$reg[rs1] \text{ op } reg[rs2]$ (for branch condition) Update pc with condition
MEM	Pass
WB	Pass

3.1.6 J-type JAL

State	Action
IF	Instruction Fetch $PC \leftarrow PC + 4$
ID	Instruction Decode $alu_reg \leftarrow PC + imm$ $reg_file[rd] \leftarrow pc + 4$
EX	$pc \leftarrow alu_reg$
MEM	Pass
WB	Pass

3.1.7 J-type JALR

State	Action
IF	Instruction Fetch $PC \leftarrow PC + 4$
ID	Instruction Decode $alu_reg \leftarrow PC + imm$ $reg_file[rd] \leftarrow pc + 4$
EX	$pc \leftarrow reg_file[rs1] + imm$
MEM	Pass
WB	Pass

3.2 State Design – State Transition

Multicycle CPU에서 특정 Instruction에 필요한 Stage만 수행하기 위한 State Transition을 작성하여 설계하였다. 아래 표에 특정 Instruction의 Stage 간 Transition을 나타내었으며,

해당 Transition은 `control_unit.v`에서 `case(state)` 및 `next_state`를 Instruction type 별로 Setting하는 것을 확인할 수 있다.

Type	Instruction	IF	ID	EX	MEM	WB
R	Arithmetic	ID	EX	WB	-	IF
I	Arithmetic	ID	EX	WB	-	IF
I	Load	ID	EX	MEM	WB	IF
S	Store	ID	EX	MEM	IF	-
B	Branch	ID	EX	IF	-	-
J	JAL	ID	EX	IF	-	-
I	JALR	ID	EX	IF	-	-

3.3 State Design – Control Signal

이를 구현하기 위해, Control Unit에서는 각 Stage에 다음과 같은 Signal을 생성한다. 0 혹은 1을 가지지 않는 Multiplexer의 select는 Selection의 Definition을 작성했다.

- `ecall` 의 경우, 추가적으로 `ECALL` type instruction을 정의했으나 아래 테이블에는 추가하지 않았다.
- `2b'00` in `wb_sel` and `op2_sel` – Enumeration을 가짐에도 `2b'00` 형태의 입력을 가지는 경우는 Don't care term으로 사용된다.
- `4b'0000` in `alu_op` – 해당 Control unit에서 나오는 `alu_op` signal은 직접 ALU에 들어가지 않고, ALU Control Unit을 거쳐 들어가게 된다. 이 때, ALU Control Unit의 `alu_op_from_inst` Control signal이 Disable된 경우 ALU Control Unit의 입력 signal을 ALU로 전파한다. 아래의 표에서 `4b'0000`의 경우에 해당하며, ALU는 Instruction으로부터 결정된 Operation을 수행한다.

3.3.1 R-type Register Arithmetic

Signal	IF	ID	EX	MEM	WB
<code>inst_reg_write</code>	1	0	0	-	0
<code>data_access</code>	0	0	0	-	0
<code>mem_read</code>	1	0	0	-	0
<code>mem_write</code>	0	0	0	-	0
<code>op_reg_write</code>	0	1	0	-	0
<code>op1_regfile</code>	0	0	1	-	0
<code>op2_sel</code>	OP2_FOUR	OP2_IMM	OP2_REG	-	00
<code>alu_reg_write</code>	0	1	1	-	0
<code>alu_op_from_inst</code>	0	0	1	-	0
<code>alu_op</code>	ALU_ADD			-	0000
<code>mem_reg_write</code>	0	0	0	-	0
<code>regfile_write</code>	0	0	0	-	1
<code>wb_sel</code>	00	00	00	-	WB_ALU_REG
<code>pc_write_cond</code>	0	0	0	-	0
<code>pc_write</code>	1	0	0	-	0
<code>pc_commit</code>	0	0	0	-	1
<code>pc_from_alu_reg</code>	0	0	0	-	0

3.3.2 I-type Immediate Arithmetic

Signal	IF	ID	EX	MEM	WB
inst_reg_write	1	0	0	-	0
data_access	0	0	0	-	0
mem_read	1	0	0	-	0
mem_write	0	0	0	-	0
op_reg_write	0	1	0	-	0
op1_regfile	0	0	1	-	0
op2_sel	OP2_FOUR	OP2_IMM	OP2_REG	-	00
alu_reg_write	0	1	1	-	0
alu_op_from_inst	0	0	1	-	0
alu_op	ALU_ADD			-	0000
mem_reg_write	0	0	0	-	0
regfile_write	0	0	0	-	1
wb_sel	00	00	00	-	WB_ALU_REG
pc_write_cond	0	0	0	-	0
pc_write	1	0	0	-	0
pc_commit	0	0	0	-	1
pc_from_alu_reg	0	0	0	-	0

3.3.3 I-type Load

Signal	IF	ID	EX	MEM	WB
inst_reg_write	1	0	0	0	0
data_access	0	0	0	1	0
mem_read	1	0	0	1	0
mem_write	0	0	0	0	0
op_reg_write	0	1	0	0	0
op1_regfile	0	0	1	0	0
op2_sel	OP2_FOUR	OP2_IMM	OP2_IMM	00	00
alu_reg_write	0	1	1	0	0
alu_op_from_inst	0	0	0	0	0
alu_op	ALU_ADD			0000	0000
mem_reg_write	0	0	0	1	0
regfile_write	0	0	0	0	1
wb_sel	00	00	00	00	WB_MEM
pc_write_cond	0	0	0	0	0
pc_write	1	0	0	0	0
pc_commit	0	0	0	0	1
pc_from_alu_reg	0	0	0	0	0

3.3.4 S-type Store

Signal	IF	ID	EX	MEM	WB
inst_reg_write	1	0	0	0	-
data_access	0	0	0	1	-
mem_read	1	0	0	0	-
mem_write	0	0	0	1	-
op_reg_write	0	1	0	0	-
op1_regfile	0	0	1	0	-
op2_sel	OP2_FOUR	OP2_IMM	OP2_IMM	00	-
alu_reg_write	0	1	1	0	-
alu_op_from_inst	0	0	0	0	-
alu_op	ALU_ADD			0000	-
mem_reg_write	0	0	0	0	-
regfile_write	0	0	0	0	-
wb_sel	00	00	00	00	-
pc_write_cond	0	0	0	0	-
pc_write	1	0	0	0	-
pc_commit	0	0	1	0	-
pc_from_alu_reg	0	0	0	0	-

3.3.5 B-type Branch

Signal	IF	ID	EX	MEM	WB
inst_reg_write	1	0	0	-	-
data_access	0	0	0	-	-
mem_read	1	0	0	-	-
mem_write	0	0	0	-	-
op_reg_write	0	1	0	-	-
op1_regfile	0	0	1	-	-
op2_sel	OP2_FOUR	OP2_IMM	OP2_REG	-	-
alu_reg_write	0	1	0	-	-
alu_op_from_inst	0	0	1	-	-
alu_op	ALU_ADD			-	-
mem_reg_write	0	0	0	-	-
regfile_write	0	0	0	-	-
wb_sel	00	00	00	-	-
pc_write_cond	0	0	1	-	-
pc_write	1	0	0	-	-
pc_commit	0	0	1	-	-
pc_from_alu_reg	0	0	1	-	-

3.3.6 J-type JAL

Signal	IF	ID	EX	MEM	WB
inst_reg_write	1	0	0	-	-
data_access	0	0	0	-	-
mem_read	1	0	0	-	-
mem_write	0	0	0	-	-
op_reg_write	0	1	0	-	-
op1_regfile	0	0	0	-	-
op2_sel	OP2_FOUR	OP2_IMM	OP2_IMM	-	-
alu_reg_write	0	1	0	-	-
alu_op_from_inst	0	0	0	-	-
alu_op	ALU_ADD			-	-
mem_reg_write	0	0	0	-	-
regfile_write	0	1	0	-	-
wb_sel	00	WB_PC_NEXT_TEMP	00	-	-
pc_write_cond	0	0	0	-	-
pc_write	1	0	1	-	-
pc_commit	0	0	1	-	-
pc_from_alu_reg	0	0	1	-	-

3.3.7 I-type JALR

Signal	IF	ID	EX	MEM	WB
inst_reg_write	1	0	0	-	-
data_access	0	0	0	-	-
mem_read	1	0	0	-	-
mem_write	0	0	0	-	-
op_reg_write	0	1	0	-	-
op1_regfile	0	0	1	-	-
op2_sel	OP2_FOUR	OP2_IMM	OP2_IMM	-	-
alu_reg_write	0	1	0	-	-
alu_op_from_inst	0	0	0	-	-
alu_op	ALU_ADD			-	-
mem_reg_write	0	0	0	-	-
regfile_write	0	1	0	-	-
wb_sel	00	WB_PC_NEXT_TEMP	00	-	-
pc_write_cond	0	0	0	-	-
pc_write	1	0	1	-	-
pc_commit	0	0	1	-	-
pc_from_alu_reg	0	0	0	-	-

4 Implementation

각 베릴로그 파일에 대한 세부 설명은 다음과 같다.

4.1 top.v – Top module

cpu.v에서 구현한 cpu 모듈을 사용하여 Multi cycle CPU를 구현하였다.

4.2 `cpu.v` – 내부 Module을 연결하여 Multicycle CPU 구성

`cpu.v`는 다음과 같은 submodule을 연결하여 구성하였다.

- `pc pc_mod` – Program counter을 구현하였다.
- `RegisterFile reg_file` – Register file으로, 값의 Read는 Asynchronous하게, Write는 control signal에 따라 Synchronous하게 작성할 수 있다.
- `operand_register op_reg` – Register file의 Output을 ALU로 전달하기 위한 레지스터인 A 와 B를 관리하며, Clock cycle에 Synchronous하게 값을 쓸 수 있다.
- `mux32bit_4x1 reg_file_write_data_mux` – Register file에 들어갈 수 있는 값을 선택하기 위한 Multiplexer를 구현하였다.
- `Memory memory` – Data memory와 Instruction memory가 통합된 하나의 Memory를 사용하고 있으며, Read 및 Write 모두 Clock signal에 Synchronous하게 접근한다.
- `memory_data_register mem_data_reg` – Memory에서 가져온 Data를 임시로 저장하기 위한 레지스터인 MDR에 Clock signal에 Synchronous하게 값을 작성한다.
- `mux32bit_2x1 memory_addr_mux` – 통합된 Memory에서, Data 영역에 접근할 지 Instruction 영역에 접근할 지 선택하도록 Multiplexer를 구현하였다.
- `control_unit ctrl_unit` – CPU의 Microsequence를 Instruction에 따라 관리하며, Microsequence의 진행과 각 Stage 별 Control signal을 조절한다.
- `instruction_register inst_reg` – Memory에서 가져온 Instruction을 임시로 저장하기 위한 레지스터인 IR에 Clock signal에 Synchronous하게 작성한다.
- `ecall_unit ec_unit` – `ecall`을 처리하기 위한 모듈을 구현하였다.
- `immediate_generator imm_gen` – 각 Instruction type에 해당하는 Immediate value를 생성한다.
- `alu_control_unit alu_ctrl_unit` – Instruction으로부터 ALU의 Opcode를 획득한다.
- `alu alu_mod` – ALU를 구현하였다.
- `alu_register alu_reg` – ALU의 출력으로 `ALUOut` 레지스터에 Clock signal에 Synchronous하게 값을 작성한다.
- `mux32bit_2x1 alu_in_1_mux` – ALU의 `in_1` 포트의 가능한 2개의 입력 중 적절한 입력을 선택한다.
- `mux32bit_4x1 alu_in_2_mux` – ALU의 `in_2` 포트의 가능한 3개의 입력 중 적절한 입력을 선택한다.
- `mux32bit_2x1 pc_source_mux` – 다음 Program counter의 값이 될 수 있는 입력 2개의 입력 중 적절한 입력을 선택한다.

4.3 control_unit.v – 마이크로컨트롤러 구현

각 Instruction type과 State에 따라, 다음과 같은 Control signal을 생성한다. Instruction 별 해당 Signal의 생성 시기는 앞에서 설명하였다.

- `inst_reg_write` – Instruction Register IR에 Memory unit의 출력을 Write하기 위한 Control signal이다.
- `data_access` – Memory에 Access하기 위해 Data address 혹은 Instruction address를 제공받는 Multiplexer의 Selection이다.
- `mem_read` – Memory로부터 값을 Read할 때 활성화되는 Control signal이다.
- `mem_write` – Memory에 값을 Write할 때 활성화되는 Control signal이다.
- `op_reg_write` – Register file의 출력에 있는 temporal register인 A, B에 Write하기 위한 Control signal이다.
- `op1_regfile` – ALU의 op1 입력으로 (현재 Program counter) / (A)를 선택하는 Multiplexer의 Selection이다.
- `op2_sel` – ALU의 op2 입력으로 (B) / (4) / (Immediate generator output)을 선택하는 Multiplexer의 Selection이다.
- `is_ecall` – 현재 Instruction이 `ecall`인 경우 활성화되는 Control signal이다.
- `alu_reg_write` – ALU result를 ALUOut에 Write하기 위한 Control signal이다.
- `alu_op_from_inst` – ALU Control Unit이 Instruction으로부터 ALU Operation을 결정하도록 하는 Control signal이다.
- `alu_op` – ALU가 수행할 Operation에 대한 코드이다. (`alu_def.v` 참조)
- `mem_reg_write` – Data Register MDR에 Memory unit의 출력을 Write하기 위한 Control signal을 구상하였으나, 해당 레지스터에는 조건 없이 Clock signal에 Synchronous 하게 값을 작성한다.
- `regfile_write` – Register file에 값을 Write하기 위한 Control signal이다.
- `wb_sel` – Register file에 Write back될 값으로 (ALUOut) / (MDR) / (PC + 4)를 선택하는 Multiplexer의 Selection이다.
- `pc_write_cond` – Program counter를 업데이트하기 위한 Branch instruction에 활성화되는 입력으로, ALU의 result에 따라 Branch taken / Branch not taken을 판단한다.
- `pc_write` – JAL 및 JALR 의 Jump instruction을 통한 갱신 및 PC + 4로 갱신하는 Stage에서 활성화되는 Control signal이다.
- `pc_commit` – 다음 Program counter의 실제 업데이트를 진행하기 위한 Control signal이다.
- `pc_from_alu_reg` – Program counter의 Source로 (현재 ALU의 Result) / (ALUOut)를 선택하는 Multiplexer의 Selection이다.

4.3.1 control_unit_def.v – 마이크로컨트롤러 상수 정의

다음과 같은 종류들의 상수가 정의되어 있다.

- CTRL_OP2_xxx – ALU의 in_2 입력을 결정하기 위한 4x1 MUX의 Selection 상수이다.
- CTRL_xx_STAGE – Multicycle CPU의 각 Stage를 내부적으로 나타내기 위한 상수이다.
- CTRL_WB_xxx – Register file의 Write back을 진행할 때, 4x1 MUX의 Selection 상수이다.

4.4 pc.v – Program counter 구현

Clock signal에 Synchronous하게 작동하며, 입력으로 들어온 현재의 Program counter, 혹은 ALU의 연산 결과로부터 Program counter의 값을 갱신한다. pc module은 다음과 같은 2개의 Control unit의 Signal에 따라 작동한다.

- pc_write – 모든 Instruction Fetch Stage와 JAL 및 JALR에서 Program counter 값의 전달 및 저장을 위한 신호이다.
- pc_commit – 현재 Program counter의 값을 갱신하기 위한 신호이다.

해당 pc module은 2개의 출력을 가지며, 다음과 같다.

- current_pc – 현재 Program counter의 값을 출력한다.
- next_temp_pc – JAL 및 JALR에서 Program counter를 레지스터 파일에 직접 전달하기 위해 추가적인 Wire를 구성하였다.

두 개의 독립적인 신호를 입력받아 Program counter를 갱신함으로써, JAL 및 JALR instruction에서 사용되는 Cycle의 수를 줄일 수 있었다. pc_write을 통해 pc_next_temp_pc로 다음 Program counter를 미리 계산하여, Register file의 입력으로 미리 계산된 Program counter를 선택하여 Write back을 진행할 수 있다. 미리 Register file에 값을 Write back 하고, 해당 값을 pc_commit을 사용하여 Program counter에 반영한다.

4.5 Memory.v – 통합된 Memory 구현

Single-cycle CPU와 다르게, Multicycle CPU에서는 Instruction memory 및 Data memory가 통합된 형태로 사용되고 있다. Instruction 영역 혹은 Data 영역으로의 접근은 addr에 연결된 Multiplexer에서 선택하여 접근 가능하다. Clock signal에 Synchronous하게 mem_read, mem_write의 신호가 주어졌을 때 각각 읽고 쓰기가 가능하다. Memory의 출력은 memory_dout을 통해 전달되며, 이는 아래의 mem_data_reg 과 inst_reg 와 연결되어 값을 Clock signal에 Synchronous하게 전달한다.

4.5.1 memory_data_register.v – Data memory register 구현

Memory unit에서 memory를 읽을 때 마다 memory_dout으로 전달되고, mem_data_reg가 Clock signal에 Synchronous하게 MDR에 값을 전달한다.

4.5.2 instruction_register.v – Instruction memory register 구현

Memory unit에서 memory를 읽을 때 memory_dout으로 값이 전달되고, Instruction 영역을 읽는 경우엔 inst_reg의 ir_write에 연결된 Control signal이 enable되어 Clock signal에 Synchronous하게 IR으로 Instruction을 전달한다.

4.6 RegisterFile.v – Register File 구현

Single-cycle CPU와 동일한 구현 형태를 가지고 있다. Clock signal에 Synchronous하게 값을 작성할 수 있으며, Multicycle CPU에서는 Write Back stage에서 작성되도록 Control signal을 설계하였다. Hard-wired zero Register인 `reg_file[0]`에 작성 시 값이 변조되지 않도록 추가적인 조건을 추가하였다. `x17` Register는 `ecall` 처리 후 `is_halted`을 설정하는데 사용되게 되며, 이를 위해 추가적으로 해당 `x17` 레지스터만을 출력하는 wire를 가지고 있다.

4.6.1 operand_register.v – Operand temporaries 구현

Register file에서 얻은 Register value는 Skeleton code에서 주어진 temporary register인 `A`와 `B`으로 값이 전달된다. Clock signal에 Synchronous하게 값을 전달하게 되며, `A` 및 `B`는 ALU의 입력을 결정하는 Multiplexer와 연결되어 Control signal에 따라 ALU의 피연산자가 될 수 있다.

4.7 immediate_generator.v – Immediate generator 구현

Single-cycle CPU에서 사용한 Immediate generator와 구조가 동일하다. Instruction이 inst로 주어질 때 `opcode`, `func3` 값을 통해 immediate value를 생성하여 `imm`으로 출력하도록 구현하였다. 해당 동작은 Clock signal에 Asynchronous하게 작동한다.

4.8 alu.v – ALU 동작 구현

Single-cycle CPU에서 사용한 ALU와 구조가 동일하며, Clock signal에 Asynchronous하게 동작하도록 구현하였다. 연산을 선택하기 위한 `alu_op`에 따라 두 개의 32bit 입력에 해당하는 연산 결과를 `alu_result`에 반환한다. 산술 연산의 경우 32bit를 모두 사용하여 결과 값을 전달하였으며, 논리 연산의 경우 LSB 1bit에 해당 논리 연산의 결과를 반환하였다.

4.8.1 alu_def.v – ALU 구성을 위한 상수 정의

기존 Single-cycle CPU에서 정의한 다음과 같은 `alu_op`를 사용하였다.

Opcode	Definition	Description
4'b0000	ALU_ADD	Add
4'b0001	ALU_SUB	Subtract
4'b0010	ALU_SLL	Logical shift left
4'b0011	ALU_SLT	Signed less than
4'b0100	ALU_SLTU	Unsigned less than
4'b0101	ALU_XOR	Bitwise XOR
4'b0110	ALU_SRL	Logical shift right
4'b0111	ALU_SRA	Arithmetic shift right
4'b1000	ALU_OR	Bitwise OR
4'b1001	ALU_AND	Bitwise AND
4'b1010	ALU_EQ	Equal
4'b1011	ALU_NE	Not equal
4'b1011	ALU_GE	Signed greater than or equal
4'b1011	ALU_GEU	Unsigned greater than or equal
4'b1111	ALU_ERR	Error signal (For debug)

4.9 alu_register.v – ALUOut을 위한 Clock 확보

ALU의 출력을 Clock signal에 Synchronous하게 Skeleton code의 ALUOut에 전달한다. 현재 Cycle의 ALU result를 생성할 때 해당 ALUOut은 직전 Cycle의 ALU result를 저장하게 된다. 두 Output을 Control signal로 선택하여 Program counter를 갱신하거나 Memory를 작성한다.

4.10 alu_control_unit.v – Instruction과 ALU Opcode 대응

Single-cycle CPU에서 구현한 ALU Control Unit과 구현이 동일하다. 입력받은 Instruction에서 요하는 ALU의 기능에 해당하는 ALU opcode를 반환하기 위한 ALU Control Unit이며, 각 Instruction 타입에 따라 ALU에 입력되는 opcode가 달라지도록 설계했다. 기본적으로 Clock signal에 Asynchronous한 로직으로 동작하지만, Control unit에서 State별 Signal이 들어올 때만 작동한다.

- Arithmetic instruction – 각 Instruction에 해당하는 Arithmetic operation으로 alu_op를 결정한다.
- Branch – Branch condition에 해당하는 논리 연산을 진행하도록 alu_op를 결정한다.
- Load / Store – ALU_ADD를 사용하여 주어진 Offset으로 Program Counter를 갱신하도록 한다.
- 이외의 경우 – ALU_ERR를 사용해 Debugging을 진행하였다.

4.11 ecall_unit.v – ecall 처리를 위한 추가 모듈 구성

x17 Register 및 ecall Instruction의 Opcode를 전달하여 프로그램이 종료되었을 경우에 is_halted를 Enable한다.

4.12 opcodes.v – CPU Instruction set에 대응되는 상수 정의

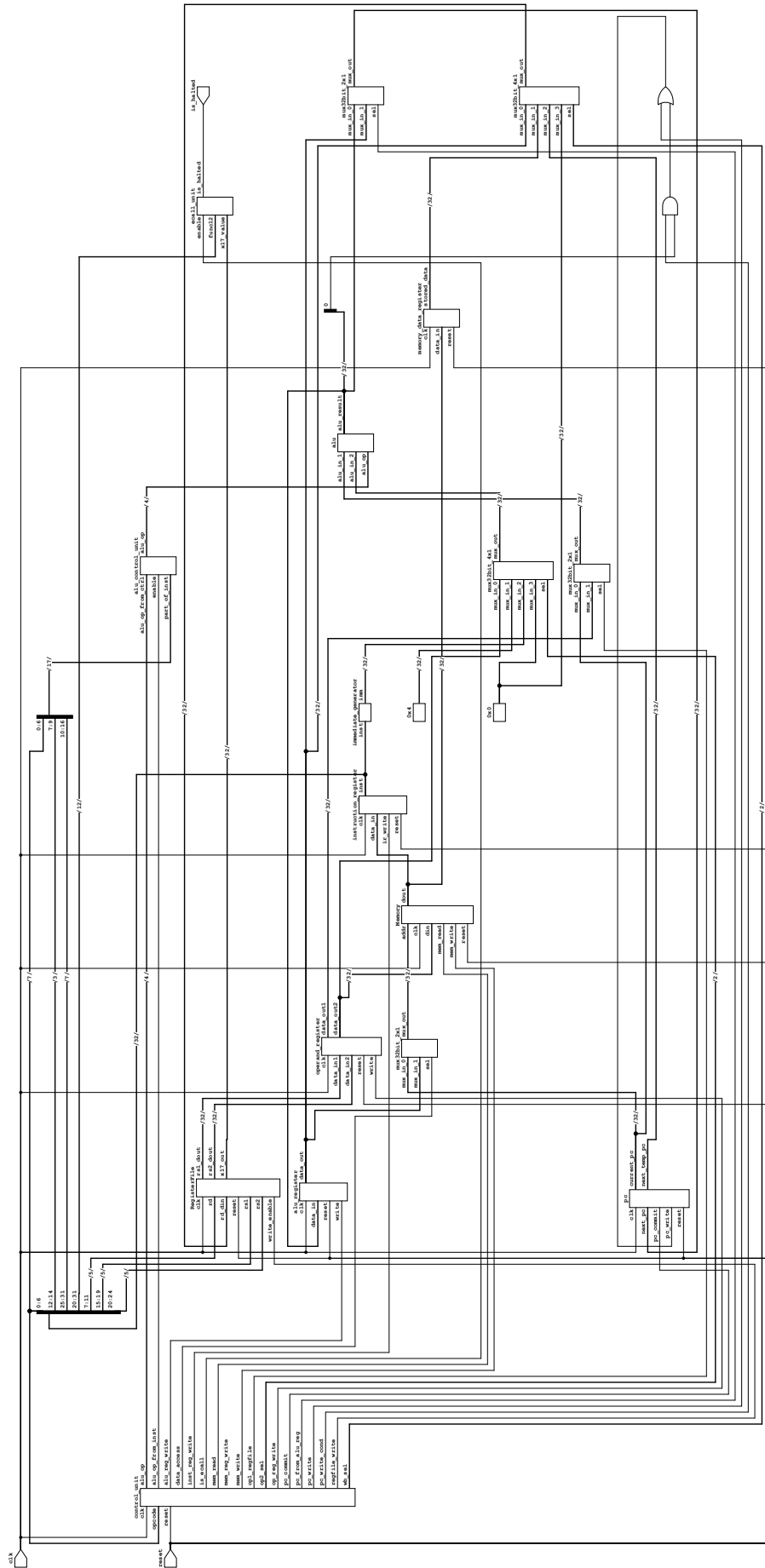
Instruction의 opcode 부분을 판단하여 operation을 특정하기 위한 상수이며, Single-cycle CPU에서 사용하던 정의와 동일하다. R-type 혹은 I-type instruction의 경우 func3 및 func7 부분에 대응되는 operation을 정의하여 사용하였다.

4.13 mux32bit_2x1.v 및 mux32bit_4x1.v – Multiplexer 구현

32bit Signal들의 입력으로부터 출력을 선택하는 Multiplexer를 구현하여 사용하였다. Clock signal과 Asynchronous하게 작동한다.

5 Schematic

최종적으로 구현한 Multicycle CPU의 Schematic은 다음과 같다.



6 Discussion

6.1 Single-cycle CPU vs. Multi-cycle CPU

Multicycle CPU와 Singlecycle CPU의 차이는 다음과 같다.

- Staged – 각 Unit의 재사용을 염두에 두고 Instruction의 실행을 Stage별로 구분하며, 이를 통해 필요 없는 Stage를 건너뛰는 방식으로 낭비되는 Cycle을 줄일 수 있다.
- Control Unit – Stage Transition을 Control Unit에서 관리하기 때문에, 각 Instruction 별로 해당하는 Stage transition의 FSM 정보를 Control Unit이 저장하도록 구현된다.
- Resource reuse – $PC + 4$ 혹은 $PC + imm$ 등을 위해 추가적으로 Adder를 구성하지 않고, Control signal으로 Stage를 구분하여 겹치는 Stage가 없도록 ALU를 재사용한다. Memory 또한 Instruction memory와 Data memory를 구분하지 않고, 하나의 Memory unit에서 각기 다른 address로의 접근을 Multiplexer를 사용해 Control하는 방식으로 구현된다.

6.2 Instruction Count of testbenches

Control Unit의 Instruction Fetch Stage에서 Instruction의 type을 $\$display$ 를 통해 출력하고, Instruction의 개수를 세었다.

Testbench	Arith.	Arith. Imm.	Load	Store	Branch	JAL	JALR
basic_mem	5	11	6	5	0	0	0
ifelse_mem	0	14	9	6	4	0	0
loop_mem	20	37	95	45	22	2	0
non-controlflow_mem	11	18	4	4	0	0	0
recursive_mem	23	332	228	168	41	62	41

앞서 서술한 장점 중 Microsequence를 통해 Clock cycle을 줄일 수 있다는 점이 Multicycle CPU의 장점이다. 현 Multicycle CPU의 구현에서 사용하는 Cycle 개수는 다음과 같다.

- Load – 5 Cycle
- Arith., Arith. Imm., Store – 4 Cycle
- Branch, JAL, JALR – 3 Cycle

Single Cycle CPU에서는 Instruction 종류에 관계 없이, Instruction Fetch부터 Write Back까지의 5 Cycle을 요하게 된다. 각 Instruction별 Cycle 수를 곱하여 Speedup을 비교해보면 다음과 같다.

Testbench	Single-cycle CPU	Multi-cycle CPU	Speedup
basic_mem	135	114	1.184
ifelse_mem	165	137	1.204
loop_mem	1105	955	1.157
non-controlflow_mem	185	152	1.217
recursive_mem	4475	3664	1.221

최대 약 1.2배의 Speedup을 확인할 수 있었다.

7 Conclusion

주어진 testbench 파일들이 테스트를 통과함을 확인할 수 있었다.
basic_mem.txt에 대한 실행 결과는 다음과 같다.

```
### SIMULATING ###
TEST END
SIM TIME : 234
TOTAL CYCLE : 116 (Answer : 116)
FINAL REGISTER OUTPUT
0 00000000
1 00000000
2 00002ffc
3 00000000
4 00000000
5 00000000
6 00000000
7 00000000
8 00000000
9 00000000
10 00000013
11 00000003
12 ffffffff7d
13 00000037
14 00000013
15 00000026
16 0000001e
17 0000000a
18 00000000
19 00000000
20 00000000
21 00000000
22 00000000
23 00000000
24 00000000
25 00000000
26 00000000
27 00000000
28 00000000
29 00000000
30 00000000
31 00000000
Correct output : 32/32
```

ifelse_mem.txt에 대한 실행 결과는 다음과 같다.

```
### SIMULATING ###
TEST END
SIM TIME : 280
TOTAL CYCLE : 139 (Answer : 139)
FINAL REGISTER OUTPUT
```



```
0 00000000
1 00000000
2 00002ffc
3 00000000
4 00000000
5 00000000
6 00000000
7 00000000
8 00000000
9 00000000
10 00000000
11 00000000
12 00000000
13 00000000
14 0000000a
15 00000028
16 00000000
17 0000000a
18 00000000
19 00000000
20 00000000
21 00000000
22 00000000
23 00000000
24 00000000
25 00000000
26 00000000
27 00000000
28 00000000
29 00000000
30 00000000
31 00000000
```

Correct output : 32/32

loop_mem.txt에 대한 실행 결과는 다음과 같다.

```
### SIMULATING ###
TEST END
SIM TIME : 1916
TOTAL CYCLE : 957 (Answer : 977)
FINAL REGISTER OUTPUT
0 00000000
1 00000000
2 00002ffc
3 00000000
4 00000000
5 00000000
6 00000000
7 00000000
8 00000000
```

```
9 00000000
10 00000000
11 00000000
12 00000000
13 00000000
14 0000000a
15 00000009
16 0000005a
17 0000000a
18 00000000
19 00000000
20 00000000
21 00000000
22 00000000
23 00000000
24 00000000
25 00000000
26 00000000
27 00000000
28 00000000
29 00000000
30 00000000
31 00000000
Correct output : 32/32
```

non-controlflow_mem.txt에 대한 실행 결과는 다음과 같다.

```
### SIMULATING ###
TEST END
SIM TIME : 316
TOTAL CYCLE : 157 (Answer : 157)
FINAL REGISTER OUTPUT
0 00000000
1 00000000
2 00002ffc
3 00000000
4 00000000
5 00000000
6 00000000
7 00000000
8 00000000
9 00000000
10 0000000a
11 0000003f
12 ffffffff1
13 0000002f
14 0000000e
15 00000021
16 0000000a
17 0000000a
```

```
18 00000000
19 00000000
20 00000000
21 00000000
22 00000000
23 00000000
24 00000000
25 00000000
26 00000000
27 00000000
28 00000000
29 00000000
30 00000000
31 00000000
Correct output : 32/32
```

recursive_mem.txt에 대한 실행 결과는 다음과 같다.

```
### SIMULATING ###
TEST END
SIM TIME : 7334
TOTAL CYCLE : 3666 (Answer : 3686)
FINAL REGISTER OUTPUT
0 00000000
1 00000000
2 00002ffc
3 00000000
4 00000000
5 00000000
6 00000000
7 00000000
8 00000000
9 00000000
10 0000000d
11 00000000
12 00000000
13 00000000
14 00000001
15 0000000d
16 00000015
17 0000000a
18 00000000
19 00000000
20 00000000
21 00000022
22 00000000
23 00000037
24 00000059
25 00000000
26 00000000
```

```
27 00000000
```

```
28 00000000
```

```
29 00000000
```

```
30 00000000
```

```
31 00000000
```

```
Correct output : 32/32
```

8 Appendix – Control Unit Code

Evaluation Criteria에서 요구한 Control Unit Code의 스크린샷을 분할하여 첨부했다. (22page 좌) - (22page 우) - (23page) 순서로 `control_unit.v`를 구성한다.

```

1 'include "alu_def.v"
2 'include "opcodes.v"
3 'include "control_unit_def.v"
4
5 module control_unit (
6     input clk,
7     input reset,
8
9     input [6:0] opcode,
10
11     output reg inst_reg_write,
12     output reg data_access,
13     output reg mem_read,
14     output reg mem_write,
15     output reg op_reg_write,
16     output reg op1_regfile,
17     output reg [1:0] op2_sel,
18     output reg is_ecall,
19     output reg alu_reg_write,
20     output reg alu_op_from_inst,
21     output reg [3:0] alu_op,
22     output reg mem_reg_write,
23     output reg regfile_write,
24     output reg [1:0] wb_sel,
25     output reg pc_write_cond,
26     output reg pc_write,
27     output reg pc_commit,
28     output reg pc_from_alu_reg
29 );
30 reg [2:0] state;
31 reg [2:0] next_state;
32
33 always @(*) begin
34     case (state)
35         'CTRL_IF_STAGE: begin
36             inst_reg_write = 1;
37             data_access = 0;
38             mem_read = 1;
39             mem_write = 0;
40             op_reg_write = 0;
41             op1_regfile = 0;
42             op2_sel = 'CTRL_OP2_FOUR;
43             is_ecall = 0;
44             alu_reg_write = 0;
45             alu_op_from_inst = 0;
46             alu_op = 'ALU_ADD;
47             mem_reg_write = 0;
48             regfile_write = 0;
49             wb_sel = 2'b0;
50             pc_write_cond = 0;
51             pc_write = 1;
52             pc_commit = 0;
53             pc_from_alu_reg = 0;
54
55             next_state = 'CTRL_ID_STAGE;
56         end
57         'CTRL_ID_STAGE: begin
58             inst_reg_write = 0;
59             data_access = 0;
60             mem_read = 0;
61             mem_write = 0;
62             op_reg_write = 1;
63             op1_regfile = 0;
64             op2_sel = 'CTRL_OP2_IMM;
65             is_ecall = opcode == 'ECALL;
66             alu_reg_write = 1;
67             alu_op_from_inst = 0;
68             alu_op = 'ALU_ADD;
69             mem_reg_write = 0;
70             pc_write_cond = 0;
71             pc_write = 0;
72             pc_commit = 0;
73             pc_from_alu_reg = 0;
74
75             if (opcode == 'JAL || opcode == 'JALR) begin
76                 regfile_write = 1;
77                 wb_sel = 'CTRL_WB_PC_NEXT_TEMP;
78             end else begin
79                 regfile_write = 0;
80                 wb_sel = 2'b0;
81             end
82
83             next_state = 'CTRL_EX_STAGE;
84         end
85     end

```

```

1 'CTRL_EX_STAGE: begin
2     inst_reg_write = 0;
3     data_access = 0;
4     mem_read = 0;
5     mem_write = 0;
6     op_reg_write = 0;
7     is_ecall = opcode == 'ECALL;
8     mem_reg_write = 0;
9     regfile_write = 0;
10    wb_sel = 2'b0;
11
12    case (opcode)
13        'ARITHMETIC: begin
14            op1_regfile = 1;
15            op2_sel = 'CTRL_OP2_REG;
16            alu_reg_write = 1;
17            alu_op_from_inst = 1;
18            alu_op = 'ALU_ADD;
19            pc_write = 0;
20            pc_commit = 0;
21            pc_write_cond = 0;
22            pc_from_alu_reg = 0;
23            next_state = 'CTRL_WB_STAGE;
24        end
25        'ARITHMETIC_IMM: begin
26            op1_regfile = 1;
27            op2_sel = 'CTRL_OP2_IMM;
28            alu_reg_write = 1;
29            alu_op_from_inst = 1;
30            alu_op = 'ALU_ADD;
31            pc_write = 0;
32            pc_commit = 0;
33            pc_write_cond = 0;
34            pc_from_alu_reg = 0;
35            next_state = 'CTRL_WB_STAGE;
36        end
37        'LOAD: begin
38            op1_regfile = 1;
39            op2_sel = 'CTRL_OP2_IMM;
40            alu_reg_write = 1;
41            alu_op_from_inst = 0;
42            alu_op = 'ALU_ADD;
43            pc_write = 0;
44            pc_commit = 0;
45            pc_write_cond = 0;
46            pc_from_alu_reg = 0;
47            next_state = 'CTRL_MEM_STAGE;
48        end
49        'JALR: begin
50            op1_regfile = 1;
51            op2_sel = 'CTRL_OP2_IMM;
52            alu_reg_write = 0;
53            alu_op_from_inst = 0;
54            alu_op = 'ALU_ADD;
55            pc_write = 1;
56            pc_commit = 1;
57            pc_write_cond = 0;
58            pc_from_alu_reg = 0;
59            next_state = 'CTRL_IF_STAGE;
60        end
61        'STORE: begin
62            op1_regfile = 1;
63            op2_sel = 'CTRL_OP2_IMM;
64            alu_reg_write = 1;
65            alu_op_from_inst = 0;
66            alu_op = 'ALU_ADD;
67            pc_write = 0;
68            pc_commit = 1;
69            pc_write_cond = 0;
70            pc_from_alu_reg = 0;
71            next_state = 'CTRL_MEM_STAGE;
72        end
73        'BRANCH: begin
74            op1_regfile = 1;
75            op2_sel = 'CTRL_OP2_REG;
76            alu_reg_write = 0;
77            alu_op_from_inst = 1;
78            alu_op = 'ALU_ADD;
79            pc_write = 0;
80            pc_commit = 1;
81            pc_write_cond = 1;
82            pc_from_alu_reg = 1;
83            next_state = 'CTRL_IF_STAGE;
84        end
85        'JAL: begin
86            op1_regfile = 0;
87            op2_sel = 'CTRL_OP2_IMM;
88            alu_reg_write = 0;
89            alu_op_from_inst = 0;
90            alu_op = 4'b0;
91            pc_write = 1;
92            pc_commit = 1;
93            pc_write_cond = 0;
94            pc_from_alu_reg = 1;
95            next_state = 'CTRL_IF_STAGE;
96        end
97        'ECALL: begin
98            op1_regfile = 0;
99            op2_sel = 2'b0;
100            alu_reg_write = 0;
101            alu_op_from_inst = 0;
102            alu_op = 4'b0;
103            pc_write = 0;
104            pc_commit = 1;
105            pc_write_cond = 0;
106            pc_from_alu_reg = 0;
107            next_state = 'CTRL_IF_STAGE;
108        end
109        default: begin
110            op1_regfile = 0;
111            op2_sel = 2'b0;
112            alu_reg_write = 0;
113            alu_op_from_inst = 0;
114            alu_op = 4'b0;
115            pc_write = 0;
116            pc_commit = 0;
117            pc_write_cond = 0;
118            pc_from_alu_reg = 0;
119            next_state = 'CTRL_ID_STAGE;
120        end
121    endcase
122 end

```

```

1  'CTRL_MEM_STAGE: begin
2      inst_reg_write = 0;
3      data_access = 1;
4      mem_read = opcode == 'LOAD;
5      mem_write = opcode == 'STORE;
6      op_reg_write = 0;
7      op1_regfile = 0;
8      op2_sel = 2'b0;
9      is_ecall = opcode == 'ECALL;
10     alu_reg_write = 0;
11     alu_op_from_inst = 0;
12     alu_op = 4'b0;
13     mem_reg_write = opcode == 'LOAD;
14     regfile_write = 0;
15     wb_sel = 2'b0;
16     pc_write_cond = 0;
17     pc_write = 0;
18     pc_commit = 0;
19     pc_from_alu_reg = 0;
20
21     if (opcode == 'STORE) next_state = 'CTRL_IF_STAGE;
22     else next_state = 'CTRL_WB_STAGE;
23 end
24
25 'CTRL_WB_STAGE: begin
26     inst_reg_write = 0;
27     data_access = 0;
28     mem_read = 0;
29     mem_write = 0;
30     op_reg_write = 0;
31     op1_regfile = 0;
32     op2_sel = 2'b0;
33     is_ecall = opcode == 'ECALL;
34     alu_reg_write = 0;
35     alu_op_from_inst = 0;
36     alu_op = 4'b0;
37     mem_reg_write = 0;
38     pc_write_cond = 0;
39     pc_write = 0;
40     pc_commit = 1;
41     pc_from_alu_reg = 0;
42
43     case (opcode)
44         'ARITHMETIC: begin
45             regfile_write = 1;
46             wb_sel = 'CTRL_WB_ALU_REG;
47         end
48         'ARITHMETIC_IMM: begin
49             regfile_write = 1;
50             wb_sel = 'CTRL_WB_ALU_REG;
51         end
52         'LOAD: begin
53             regfile_write = 1;
54             wb_sel = 'CTRL_WB_MEM;
55         end
56         default: begin
57             regfile_write = 0;
58             wb_sel = 2'b0;
59         end
60     endcase
61
62     next_state = 'CTRL_IF_STAGE;
63 end
64
65 default: begin
66     inst_reg_write = 0;
67     data_access = 0;
68     mem_read = 0;
69     mem_write = 0;
70     op_reg_write = 0;
71     op1_regfile = 0;
72     op2_sel = 2'b0;
73     is_ecall = 0;
74     alu_reg_write = 0;
75     alu_op_from_inst = 0;
76     alu_op = 4'b0;
77     mem_reg_write = 0;
78     regfile_write = 0;
79     wb_sel = 2'b0;
80     pc_write_cond = 0;
81     pc_write = 0;
82     pc_from_alu_reg = 0;
83
84     next_state = 'CTRL_IF_STAGE;
85 end
86 endcase
87 end
88
89 always @(posedge clk) begin
90     if (reset) begin
91         state <= 'CTRL_IF_STAGE;
92     end else begin
93         state <= next_state;
94     end
95 end
96 endmodule

```