Custom Microcontroller Specification (Ver 1.1)

Duo Lu <duolu@asu.edu>

For ASU CSE 320 class in Spring 2015, last update: 02/1/2020.

This project has been obsolete and a full implementation is provided in this document.

1 Outline

The microcontroller (similar to Microchip PIC12, simplified extensively, not compatible in instruction set) in this project is a 3 cycle non-pipeline 8 bit Harvard structure microcontroller, illustrated as the following block diagram. The stage group includes components working in that stage. Data path is shown as black arrows, and control signals are red arrows.

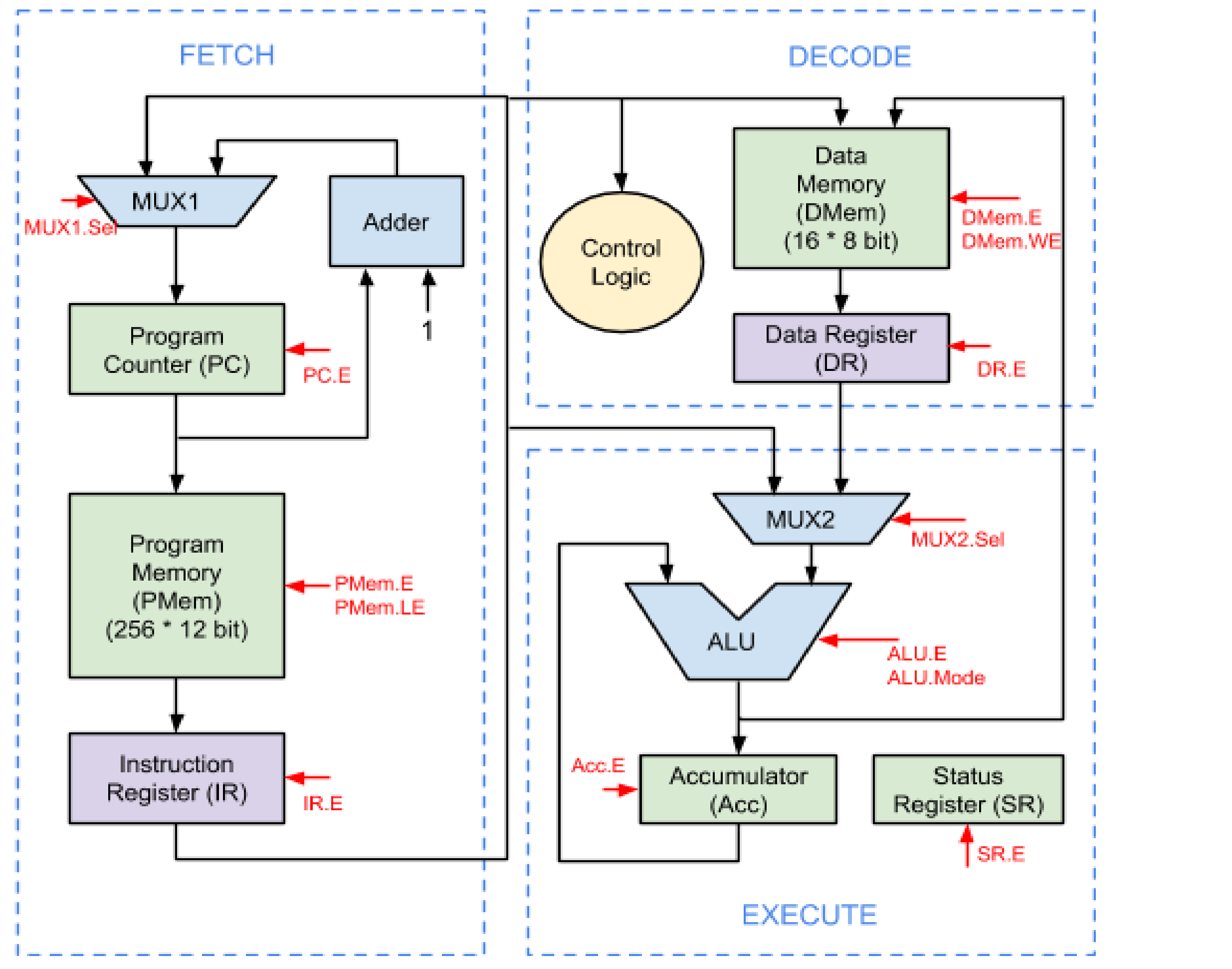


Figure 1: Block diagram of our custom microcontroller.

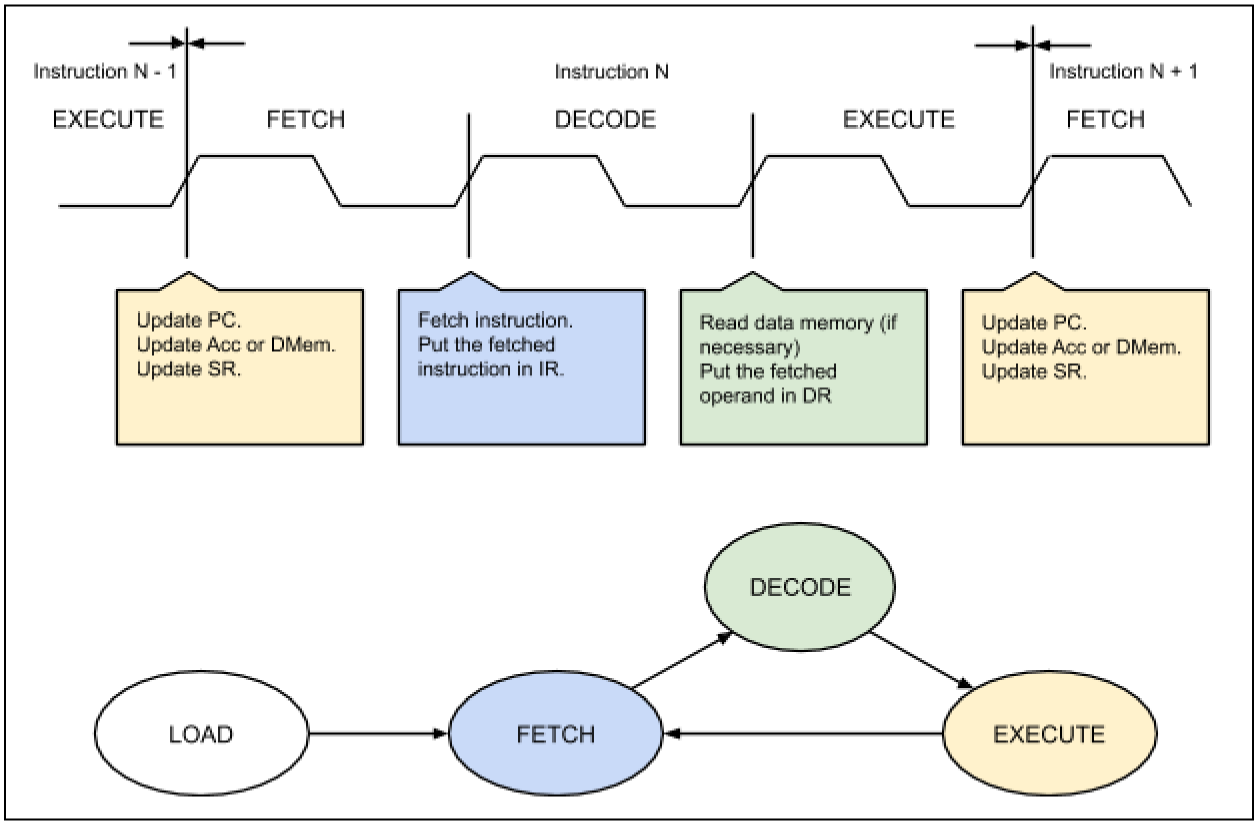
The following two types of components hold programming context.

- Program counter, program memory, data memory, accumulator, status register (green boxes). They are programmer visible registers and memories.
- Instruction register and data register (purple boxes). They are programmer invisible registers.

The following two types of components are boolean logics that do the actual computation work. They are stateless

- ALU, MUX1, MUX2, Adder (blue boxes), used as functional units.
- Control Logic (yellow box), used to denote all control signals (red signal)

The timing and state transition is shown in the following diagram.


Instruction N - 1
Instruction N
Instruction N + 1
EXECUTE
FETCH
DECODE
EXECUTE
FETCH
Update PC. Update Acc or DMem. Update SR.
Fetch instruction. Put the fetched instruction in IR.
Read data memory (if necessary) Put the fetched operand in DR
Update PC. Update Acc or DMem. Update SR.
DECODE
LOAD
FETCH
EXECUTE

Each instruction needs 3 clock cycles to finish, i.e. FETCH stage, DECODE stage, and EXECUTE stage. Note that it is not pipelined. Together with the initial LOAD state, it can be considered as a FSM of 3 states (technically 4 states).

States:

1. LOAD (initial state): load program to program memory, which takes 1 cycle per instruction loaded;
2. FETCH (first cycle): fetch current instruction from program memory;
3. DECODE (second cycle): decode instruction to generate control logic, read data memory for operand;
4. EXECUTE (of third cycle): execute instruction

Transitions:

1. LOAD ----> FETCH (initialization finish)
    a. Clear content of PC, IR, DR, Acc, SR; DMem is not required to be cleared.
2. FETCH ----> DECODE (rising edge of second cycle)
    a. IR = PMem [PC];
3. DECODE ----> EXECUTE (rising edge of third cycle)
    a. DR = DMem [IR[3:0]];
4. EXECUTE ----> FETCH (rising edge of first cycle and fourth cycle)
    a. For non-branch instruction, PC = PC + 1; for branch instruction, if branch is taken, PC = IR [7:0], otherwise PC = PC + 1;
    b. For ALU instruction, if the result destination is accumulator, Acc = ALU.Out; if the result destination is data memory, DMem [IR[3:0]] = ALU.Out.
    c. For ALU instruction, SR = ALU.Status;

The transitions can be simplified using enable port of corresponding registers, e.g. assign ALU.Out to Acc at every clock rising edge if Acc.E is set to 1. Such control signals as Acc.E are generated as a boolean function of both current state and the current instruction.

2 Components

Registers

The microcontroller has 3 programmer visible register:

1. Program Counter (8 bit, denoted as PC): contains the index of current executing instruction.
2. Accumulator (8 bit, denoted as Acc): holds result and 1 operand of the arithmetic or logic calculation.
3. Status Register (4 bit, denoted as SR): holds 4 status bit, i.e. Z, C, S, O.
    a. Z (zero flag, SR[3]): 1 if result is zero, 0 otherwise.
    b. C (carry flag, SR[2]): 1 if carry is generated, 0 otherwise.
    c. S (sign flag, SR[1]): 1 if result is negative (as 2's complement), 0 otherwise.
    d. O (overflow flag, SR[0]): 1 if result generates overflow, 0 otherwise.

Each of these registers has an enable port, as a flag for whether the value of the register should be updated in state transition. They are denoted as PC.E, Acc.E, and SR.E.

The microcontroller has 2 programmer invisible register (i.e. they cannot be manipulated by programer):

1. Instruction Register (12 bit, denoted as IR): contains the current executing instruction.
2. Data Register (8 bit, denoted as DR): contains the operand read from data memory.

Similarly, each of these registers has an enable port as a flag for whether the value of the register should be updated in state transition. They are denoted as IR.E, and DR.E.

Program memory

The microcontroller has a 256 entry program memory that stores program instructions, denoted as PMem. Each entry is 12 bits, the ith entry is denoted as PMem[i]. The program memory has the following input / output ports.

- Enable port (1 bit, input, denoted as PMem.E): enable the device, i.e. if it is 1, then the entry specified by the address port will be read out, otherwise nothing is read out.
- Address port (8 bit, input, denoted as PMem.Addr): specify which instruction entry is read out, connected to PC.
- Instruction port (12 bit, output, denoted as PMem.I): the instruction entry that is read out, connected to IR.

3 special ports are used to load programs to the memory, not used for executing instructions.

- Load enable port (1 bit, input, denoted as PMem.LE): enable the load, i.e. if it is 1, then the entry specified by the address port will be load with the value specified by the load instruction input port; otherwise, the entry specified by the address port will be read out on instruction port, and value on instruction load port is ignored.
- Load address port (8 bit, input, denoted as PMem.LA): specify which instruction entry is loaded.
- Load instruction port (12 bit, input, denoted as PMem.LI): the instruction that is loaded.

For example, if the address point is supplied with 8’b0000_0011 and enable is set to 1, the fourth entry is read out on the instruction port.

Note that program load only takes effect on clock rising edge, while instruction read out happens all the time.

Data memory

The microcontroller has a 16 entry data memory, denoted as DMem. Each entry is 8 bits, the ith entry is denoted as DMem[i]. The program memory has the following input / output ports.

- Enable port (1 bit, input, denoted as DMem.E): enable the device, i.e. if it is 1, then the entry specified by the address port will be read out or written in; otherwise nothing is read out or written in.
- Write enable port(1 bit, input, denoted as DMem.WE): enable the write, i.e. if it is 1, then the entry specified by the address port will be written with the value specified by the data input port; otherwise, the entry specified by the address port will be read out on data output port, and value on data input port is ignored.
- Address port (4 bit, input, denoted as DMem.Addr): specify which data entry is read out, connected to IR[3:0].
- Data input port (8 bit, input, denoted as DMem.DI): the value that is written in, connected to ALU.Out.
- Data output port (8 bit, output, denoted as DMem.DO): the data entry that is read out, connected to MUX2.In1.

For example, if the address point is supplied with 8’0000_0011, data input port is supplied with 8’0000_0000, enable is set to 1, and write enable is set to 1, the fourth entry of the data memory is written with value 0.

As another example, if the address point is supplied with 8’0000_0011, data input port is supplied with 8’0000_0000, enable is set to 1, while write enable is set to 0, the fourth entry of the data memory is read out on the data output port.

Note that writing only takes effect on clock rising edge, while reading happens all the time, similar to program memory.

PC adder

PC adder is used to add PC by 1, i.e. move to the next instruction. This component is pure combinational. It has the following port.

- Adder input port (8 bit, input, denoted as Adder.In): connected to PC.
- Adder output port (8 bit, output, denoted as Adder.Out): connected to MUX1.In2.

MUX1

MUX1 is used to choose the source for updating PCs. If the current instruction is not a branch or it is a branch but the branch is not taken, PC is incremented by 1; otherwise PC is set to the jumping target, i.e. IR [7:0]. It has the following port.

- MUX1 input 1 port (8 bit, input, denoted as MUX1.In1): connected to IR [7:0].
- MUX1 input 2 port (8 bit, input, denoted as MUX1.In2): connected to Adder.Out.
- MUX1 selection port (1 bit, input, denoted as MUX1.Sel): connected to control logic.
- MUX1 output port (8 bit, output, denoted as MUX1.Out): connected to PC.

ALU

ALU is used to do the actual computation for the current instruction. This component is pure combinational. It has the following port. The mode of ALU is listed in the following table.

- ALU operand 1 port (8 bit, input, denoted as ALU.Operand1): connected to Acc.
- ALU operand 1 port (8 bit, input, denoted as ALU.Operand2): connected to MUX2.Out.
- ALU enable port (1 bit, input, denoted as ALU.E): connected to control logic.
- ALU mode port (4 bit, input, denoted as ALU.Mode): connected to control logic.
- Current flags port (4 bit, input, denoted as ALU.CFlags): connected to SR.
- ALU output port (8 bit, output, denoted as ALU.Out): connected to DMem.DI.
- ALU flags port (4 bit, output, denoted as ALU.Flags): the Z (zero), C (carry), S (sign), O (overflow) bits, from MSB to LSB, connected to status register.

mode (binary)	mode (hex)	function	comments (instructions)
0000	0	Out = Operand1 + Operand2	
0001	1	Out = Operand1 - Operand2	
0010	2	Out = Operand1	for MOVAM
0011	3	Out = Operand2	for MOVMA and MOVIA
0100	4	Out = Operand1 AND Operand2	
0101	5	Out = Operand1 OR Operand2	
0110	6	Out = Operand1 XOR Operand2	
0111	7	Out = Operand2 - Operand1	
1000	8	Out = Operand2 + 1	
1001	9	Out = Operand2 - 1	
1010	A	Out = (Operand2 << Operand1 [2:0]) | (Operand2 >> 8 - Operand1 [2:0])	for ROTATEL
1011	B	Out = (Operand2 >> Operand1 [2:0]) | (Operand2 << 8 - Operand1 [2:0])	for ROTATER
1100	C	Out = Operand2 << Operand1 [2:0]	logical shift left
1101	D	Out = Operand2 >> Operand1 [2:0]	logical shift right
1110	E	Out = Operand2 >>> Operand1 [2:0]	arithmetic shift right
1111	F	Out = 0 - Operand2	2’s complement

MUX2

MUX1 is used to choose the source for operand 2 of ALU. If the current instruction is M type, operand 2 of ALU comes from data memory; if the current instruction is I type, operand 2 of ALU comes from the instruction, i.e. IR [7:0]. It has the following port.

- MUX2 input 1 port (8 bit, input, denoted as MUX2.In1): connected to IR [7:0].
- MUX2 input 2 port (8 bit, input, denoted as MUX2.In2): connected to DR.
- MUX2 selection port (1 bit, input, denoted as MUX2.Sel): connected to control logic.
- MUX2 output port (8 bit, output, denoted as MUX2.Out): connected to ALU.Operand2.

3 Instruction Set

Each instruction is 12 bits. There are 3 types of instructions by encoding, shown as following:

1. M type: one operand is the accumulator (sometimes ignored) and the other operand is from data memory; the result can be stored into the accumulator or the data memory entry (same entry as the second operand).
2. I type: one operand is the accumulator and the other operand is an immediate number encoded in instruction; the result is stored into the accumulator.
3. S type: special instruction, no operand required. (e.g. NOP)

The instruction encoding space is shown in the following table.

code space (binary)	code space (hex)	instructions	# of ins	comments
0000_0000_0000 - 0000_1111_1111	000 - 0FF	special instructions (S type)	256	Currently only NOP used, 255 free slots
0001_0000_0000 - 0001_1111_1111	100 - 1FF	unconditional jump (I type)	1	GOTO
0010_0000_0000 - 0011_1111_1111	200 - 3FF	ALU instructions (M type)	32	16 instructions, 2 destination choices each
0100_0000_0000 - 0111_1111_1111	400 - 7FF	conditional jump (I type)	4	JZ, JC, JS, JC
1000_0000_0000 - 1111_1111_1111	800 - FFF	ALU instructions (I type)	8	Currently 7 used, 1 free slot

These instructions can be grouped into 4 category by function.

1. ALU instruction: using ALU to compute result;
2. Unconditional branch: the GOTO instruction;
3. Conditional branch: the JZ, JC, JS, JO instruction;
4. Special instruction: the NOP.

M type instructions

The general format of M type instruction is shown as following.

M type	001	d	opcode (4 bits)	DMem index (4 bits)

The following table contains the detailed information of each M type instruction. Note that “aaaa” encodes the 4 bit address of data memory, and the “d” bit means destination of the result, i.e. if d = 1, result is written to Acc, otherwise the result is written to the same memory location as the operand.

Note that all M type instructions are ALU instructions.

instruction mnemonics	function	encoding (binary)	status affected	example (encoding) (assembly) (meaning)
ADD	add a memory entry with accumulator	001d_0000_aaaa	Z, C, S, O	0011_0000_0001 ADD Acc, Acc, DMem[1] (Acc = Acc + DMem[1])
SUBAM	subtract accumulator by a memory entry	001d_0001_aaaa	Z, C, S, O	0011_0001_0000 SUBAM Acc, Acc, DMem[0] (Acc = Acc - DMem[0])
MOVAM	move the value of accumulator to a memory entry	0010_0010_aaaa	none	0010_0010_0000 MOVAM DMem[0], Acc (Acc = DMem[0])
MOVMA	move the value of a memory entry to accumulator	0011_0011_aaaa	none	0011_0011_0000 MOVMA Acc, DMem[0] (DMem[0] = Acc)
AND	bitwise AND a memory entry with accumulator	001d_0100_aaaa	Z	0010_0100_0000 AND DMem[0], Acc, DMem[0] (DMem[0] = DMem[0] AND Acc)
OR	bitwise OR a memory entry with accumulator	001d_0101_aaaa	Z	0010_0101_0000 OR DMem[0], Acc, DMem[0] (DMem[0] = DMem[0] OR Acc)
XOR	bitwise XOR a memory entry with accumulator	001d_0110_aaaa	Z	0010_0110_0000 XOR DMem[0], Acc, DMem[0] (DMem[0] = DMem[0] XOR Acc)
SUBMA	subtract a memory entry by accumulator	001d_0111_aaaa	Z, C, S, O	0010_0001_0000 SUBAM DMem[0], DMem[0], Acc (DMem[0] = DMem[0] - Acc)
INC	increment a memory entry	0010_1000_aaaa	Z, C, S, O	0011_1000_0000 INC DMem[0] (DMem[0] = DMem[0] + 1)
DEC	decrement a memory entry	0010_1001_aaaa	Z, C, S, O	0011_1001_0000 DEC DMem[0] (DMem[0] = DMem[0] - 1)
ROTATEL	circulative shift left a memory entry, by the number of bits specified by accumulator	0010_1010_aaaa	none	0010_1010_0000 ROTATEL DMem[0] (see comments below)
ROTATER	circulative shift left a memory entry, by the number of bits specified by accumulator	0010_1011_aaaa	none	0010_1011_0000 ROTATER DMem[0] (see comments below)
SLL	shift a memory entry left, by the number of bits specified by accumulator	0010_1100_aaaa	Z, C	0010_1100_0000 SLL DMem[0] (see comments below)
SRL	shift a memory entry right, logical (fill 0), by the number of bits specified by accumulator	0010_1101_aaaa	Z, C	0010_1101_0000 SRL DMem[0] (see comments below)

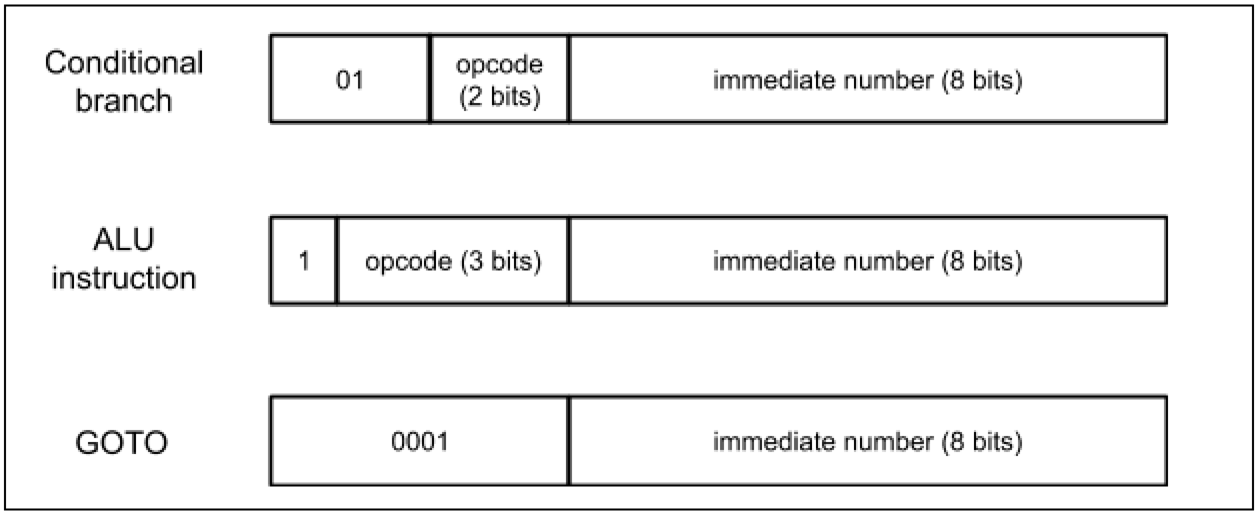
SRA	shift a memory entry right, arithmetic (fill original MSB), by the number of bits specified by accumulator	0010_1110_aaaa	Z, C, S	0010_1110_0000 SRA DMem[0] (see comments below)
COMP	take 2's complement of a memory entry, i.e. 0 subtracted by the memory entry	0010_1111_aaaa	Z, C, S, O	0010_1111_0000 COMP DMem[0] (DMem[0] = - DMem[0])

Comments:

1. For circulative shift, only the operand from memory entry is rotated, i.e. flags are not involved. For example, if 8'b1000_0000 is shifted left circulative for 1 bit, it becomes 8'b0000_0001, and no flag is affected.
2. For logical shift instructions, 0 is always filled in, and C flag is set to the last bit that is shifted out. For example, if 8'b1000_0000 is shifted left logically for 1 bit, it becomes 8'b0000_0000, and C = 1. Another example, if 8'b0000_0100 is shifted right logically for 3 bit, it becomes 8'b0000_0000, and C = 1. If no bit is shifted out, Z flag and C flag are not affected.
3. For arithmetic shift right instructions, the MSB is filled in, and C flag is set to the last bit that is shifted out. For example, if 8'b1000_0001 is shifted right arithmetically for 1 bit, it becomes 8'b1100_0000, and C = 1. Another example, if 8'b0000_0100 is shifted right arithmetically for 3 bit, it becomes 8'b0000_0000, and C = 1. If no bit is shifted out, Z flag and C flag are not affected.

I type instructions

The general format of I type instruction is shown as follows.


Conditional branch
01
opcode (2 bits)
immediate number (8 bits)
ALU instruction
1
opcode (3 bits)
immediate number (8 bits)
GOTO
0001
immediate number (8 bits)

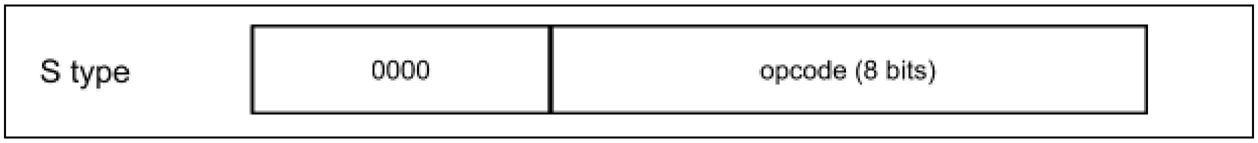
The following table contains the detailed information of each I type instruction.

Note that I type instructions contain unconditional branch, conditional branch, and ALU instructions.

instruction mnemonics	function	encoding	status affected	example (encoding) (assembly) (meaning)
GOTO	unconditional branch	0001_xxxx_xxxx	none	0001_0000_0111 GOTO 7 (goto the 8th instruction)
JO	jump to the instruction indexed by the immediate number, if Z flag is 1	0100_xxxx_xxxx	none	0100_0000_0111 JO 7 (goto the 8th instruction if SR[3] == 1)
JS	jump to the instruction indexed by the immediate number, if C flag is 1	0101_xxxx_xxxx	none	0101_0000_0111 JS 7 (goto the 8th instruction if SR[2] == 1)
JC	jump to the instruction indexed by the immediate number, if S flag is 1	0110_xxxx_xxxx	none	0110_0000_0111 JC 7 (goto the 8th instruction if SR[1] == 1)
JZ	jump to the instruction indexed by the immediate number, if O flag is 1	0111_xxxx_xxxx	none	0111_0000_0111 JZ 7 (goto the 8th instruction if SR[0] == 1)
ADDI	add accumulator with immediate number	1000_xxxx_xxxx	Z, C, S, O	1000_0000_1000 ADDI Acc, Acc, 8 (Acc = Acc + 8)
SUBAI	subtract accumulator by immediate number	1001_xxxx_xxxx	Z, C, S, O	1001_0000_1000 SUBAI Acc, Acc, 8 (Acc = Acc - 8)
RSV	(reserved, do nothing) (actually it move the value of accumulator to accumulator)	1010_xxxx_xxxx	none	
MOVIA	move immediate number to accumulator	1011_xxxx_xxxx	none	1011_0000_0000 MOVIA Acc, 0 (Acc = 0)
ANDI	bitwise AND accumulator with immediate number	1100_xxxx_xxxx	Z	1100_0000_1111 ANDI Acc, Acc, 0x0F (Acc = Acc AND 0x0F)
ORI	bitwise OR accumulator with immediate number	1101_xxxx_xxxx	Z	1101_1111_0000 ORI Acc, Acc, 0xF0 (Acc = Acc OR 0xF0)
XORI	bitwise XOR accumulator with immediate number	1110_xxxx_xxxx	Z	1110_0000_1111 XORI Acc, Acc, 0x0F (Acc = Acc XOR 0x0F)
SUBIA	subtract accumulator by immediate number	1111_xxxx_xxxx	Z, C, S, O	1111_0000_1000 SUBIA Acc, 8, Acc (Acc = 8 - Acc)

S type instructions

The general format of S type instruction is shown as follows.



There is only one S type instruction, i.e. the NOP instruction.

instruction mnemonics	function	encoding	status affected	example
NOP	no operation	0000_0000_0000	none	NOP

Control Signal

Control signal is derived from the current state and current instruction. The control logic component is purely combinational. There are in total 12 control signals, listed as follows.

- PC.E: enable port of program counter (PC);
- Acc.E: enable port of accumulator (Acc);
- SR.E: enable port of status register (SR);
- IR.E: enable port of instruction register (IR);
- DR.E: enable port of data register (DR);
- PMem.E: enable port of program memory (PMem);
- DMem.E: enable port of data memory (DMem);
- DMem.WE: write enable port of data memory (DMem);
- ALU.E: enable port of ALU;
- ALU.Mode: mode selection port of ALU;
- MUX1.Sel: selection port of MUX1;
- MUX2.Sel: selection port of MUX2;

The following table documents the details of how these control signals are generated. important signals are marked in red.

Ins	Stage	PC. E	Acc. E	SR. E	IR. E	DR. E	PMem .E	DMem .E	DMem. WE	ALU. E	ALU. Mode	MUX 1.Sel	MUX 2.Sel
NOP	FETCH	0	0	0	1	0	1	0	0	0	x	x	x
(0000)	DECODE	0	0	0	0	0	0	0	0	0	x	x	x
	EXECUTE	1	0	0	0	0	0	0	0	0	x	1	x
GOTO	FETCH	0	0	0	1	0	1	0	0	0	x	x	x
(0001)	DECODE	0	0	0	0	0	0	0	0	0	x	x	x
	EXECUTE	1	0	0	0	0	0	0	0	0	x	0	x
ALU	FETCH	0	0	0	1	0	1	0	0	0	x	x	x
M Type	DECODE	0	0	0	0	1	0	1	0	0	x	x	x
(001x)	EXECUTE	1	$IR[8]$	1	0	0	0	$\overline{IR}[8]$	$\overline{IR}[8]$	1	$IR[7:4]$	1	1
JZ, JC, JS, JO	FETCH	0	0	0	1	0	1	0	0	0	x	x	x
(01xx)	DECODE	0	0	0	0	0	0	0	0	0	x	x	x
	EXECUTE	1	0	0	0	0	0	0	0	0	x	$\overline{SR*}$	x
ALU	FETCH	0	0	0	1	0	1	0	0	0	x	x	x
I Type	DECODE	0	0	0	0	0	0	0	0	0	x	x	x
(1xxx)	EXECUTE	1	1	1	0	0	0	0	0	1	$IR[10:8]$	1	0

Comments:

1. Note that in the EXECUTE state of conditional branch instructions, the value of MUX1.Sel is actually ~SR[IR [9:8]];
2. Note that in the EXECUTE stage of ALU I type instruction, the value of ALU.Mode is actually {0, IR[10:8]}. In the table above 0 extension is assumed.
3. Be careful that PMem.LE is not shown in this table. If the processor is not in LOAD state, PMem.LE is always set to 0.

The type and category of instruction can be identified by the first 4 bits of the instruction, i.e IR[11:8], as denoted in the first column of the above table.

Besides, when loading the program, the control signal is generated as below.

State	PMem. LE	PC. E	Acc. E	SR. E	IR. E	DR. E	PMem .E	DMem .E	DMem. WE	ALU. E	ALU. Mode	MUX 1.Sel	MUX 2.Sel
LOAD	1	0	0	0	0	0	1	0	0	0	x	x	x

If the processor is in LOAD state, PMem.LE is always set to 1.

4 Sample Testing Program

Test1: Program loading, accumulator, and memory loading, GOTO

#	instruction (binary)	instruction (assembly)	instruction (meaning)
0	0000_0000_0000	NOP	(no operation)
1	1011_0000_0001	MOVIA Acc, 1	Acc = 1
2	0010_0010_0000	MOVAM DMem[0], Acc	DMem[0] = Acc = 1
3	1011_0000_0000	MOVIA Acc, 0	Acc = 0
4	0011_0011_0000	MOVMA Acc, DMem[0]	Acc = DMem[0] = 1
5	0001_0000_0101	GOTO 5	(jump to itself, i.e. infinite loop)

Test2: Addition and subtraction (M type)

#	instruction (binary)	instruction (assembly)	instruction (meaning)
0	0000_0000_0000	NOP	(no operation)
1	1011_0000_0001	MOVIA Acc, 1	Acc = 1
2	0010_0010_0000	MOVAM DMem[0], Acc	DMem[0] = Acc = 1
3	0011_0000_0000	ADD Acc, Acc, DMem[0]	Acc = Acc + DMem[0] = 1 + 1 = 2
4	0010_0000_0000	ADD DMem[0], Acc, DMem[0];	DMem[0] = Acc + DMem[0] = 1 + 2 = 3
5	0011_0001_0000	SUBAM Acc, Acc, DMem[0];	Acc = Acc - DMem[0] = 2 - 3 = -1
6	0010_0001_0000	SUBAM DMem[0], Acc, DMem[0];	DMem[0] = Acc - DMem[0] = (-1) - 3 = -4
7	0011_0111_0000	SUBMA Acc, DMem[0], Acc;	Acc = DMem[0] - Acc = (-4) - (-1) = -3
8	0010_0111_0000	SUBMA DMem[0], DMem[0], Acc;	DMem[0] = DMem[0] - Acc = (-4) - (-3) = -1
9	0001_0000_1001	GOTO 9	(jump to itself, infinite loop)

Test3: Logic operation (M type)

#	instruction (binary)	instruction (assembly)	instruction (meaning)
0	0000_0000_0000	NOP	(no operation)
1	1011_0000_0101	MOVIA Acc, 0x05	Acc = 0x05
2	0010_0010_0000	MOVAM DMem[0], Acc	DMem[0] = Acc = 0x05
3	0010_0010_0001	MOVAM DMem[1], Acc	DMem[1] = Acc = 0x05
4	0010_0010_0010	MOVAM DMem[2], Acc	DMem[2] = Acc = 0x05
5	1011_0000_0011	MOVIA Acc, 0x03	Acc = 0x03
6	0010_0100_0000	ADD DMem[0], Acc, DMem[0]	DMem[0] = Acc AND DMem[0] = 0x03 AND 0x05 = 0x01
7	0010_0101_0001	OR DMem[1], Acc, DMem[1];	DMem[1] = Acc OR DMem[1] = 0x03 OR 0x05 = 0x07
8	0010_0110_0010	XOR DMem[2], Acc, DMem[2];	DMem[2] = Acc XOR DMem[2] = 0x03 XOR 0x05 = 0x06
9	0001_0000_1001	GOTO 9	(jump to itself, infinite loop)

Test4: Addition, subtraction, logic operation (I type)

#	instruction (binary)	instruction (assembly)	instruction (meaning)
0	0000_0000_0000	NOP	(no operation)
1	1011_0000_0001	MOVIA Acc, 1	Acc = 1
2	1010_0000_0000	RSV	(reserved instruction, do nothing)
3	1000_0000_0111	ADDI Acc, Acc, 7	Acc = Acc + 7 = 1 + 7 = 8
4	1001_0000_0110	SUBAI Acc, Acc, 6	Acc = Acc - 6 = 8 - 6 = 2
5	1111_0000_0111	SUBIA Acc, 7, Acc	Acc = 7 - Acc = 7 - 2 = 5
6	1100_0000_0011	ANDI Acc, Acc, 0x03	Acc = Acc AND 0x03 = 0x05 AND 0x03 = 0x01
7	1101_0000_0101	ORI Acc, Acc, 0x05	Acc = Acc OR 0x05 = 0x01 OR 0x05 = 0x05
8	1110_0000_0011	XOR Acc, Acc, 0x03	Acc = Acc XOR 0x03 = 0x05 XOR 0x03 = 0x06
9	0001_0000_1001	GOTO 9	(jump to itself, infinite loop)

5 Verilog Implementation

Here is an example implementation made by the author, in a single Verilog file. This implementation is briefly tested in ALDEC simulator (not strictly evaluated, so there may be bugs).

```
// program memory module
module pmem(

   input clk,

   input e,
   input [7:0] addr,
   output reg [11:0] ins,

   input le,
   input [7:0] la,
   input [11:0] li
   );


   reg   [11:0] pmem[0:255];
   integer i;

   initial begin

      for   (i = 0; i < 256; i = i + 1)
         pmem[i]   = 12'b0000_0000_0000;

   end

   always @(*) begin

      if(e == 1'b1 && le == 1'b0)
         ins <= pmem[addr];
      else
         ins <= 12'bzzzz_zzzz_zzzz;
   end


   always @(posedge clk) begin

      if(e == 1'b1 && le == 1'b1) begin
         pmem[la] <= li;
      end
   end



endmodule


// data memory module
```

```
module dmem(

   input clk,

   input e,
   input we,
   input [3:0] addr,
   input [7:0] din,
   output reg [7:0] dout

   );


   reg   [7:0] dmem[0:15];
   integer i;

   initial begin

      for   (i = 0; i < 16; i = i + 1)
         dmem[i]   = 8'b0000_0000;

   end


   always @(*) begin

      if(e == 1'b1 && we == 1'b0)
         dout <= dmem[addr];
      else
         dout <= 8'bzzzz_zzzz;
   end

   always @(posedge clk) begin

      if(e == 1'b1 && we == 1'b1) begin
         dmem[addr] <= din;
      end
   end



endmodule


// program counter adder
module pc_adder(

   input [7:0] in,
   output [7:0] out

   );

   assign out = in + 1;

endmodule


// 8 bit mux, choosing from 2 input
module mux_8b(

   input [7:0] in1,
   input [7:0] in2,
   input sel,
   output reg [7:0] out
```

```
  );

  always @(*) begin

    case(sel)
      1'b0 : out = in1;
      1'b1 : out = in2;
      default : out = 8'bzzzz_zzzz;
    endcase

  end

endmodule


// 8 bit ALU
module alu_8b(

  input [7:0] operand1,
  input [7:0] operand2,
  input e,
  input [3:0] mode,
  input [3:0] cflags,
  output reg [7:0] out,
  output reg [3:0] flags

  );

  // flag bits
  reg z;
  reg c;
  reg s;
  reg o;

  always @(*) begin

    // refer to http://teaching.idallen.com/dat2343/10f/notes/040_overflow.txt for overflow detection


    case(mode)

      // ADD, ADDI
      4'b0000 : begin
        {c, out} = operand1 + operand2;
        z = (out == 8'b0000_0000 ? 1 : 0);
        s = out[7];
        o = (~operand1[7] & ~operand2[7] & s) | (operand1[7] & operand2[7] & ~s);
        flags = {z, c, s, o};
      end

      // SUBAM, SUBAI
      4'b0001 : begin
        {c, out} = {1'b0, operand1} - {1'b0, operand2};
        z = (out == 8'b0000_0000 ? 1 : 0);
        s = out[7];
        o = (~operand1[7] & operand2[7] & s) | (operand1[7] & ~operand2[7] & ~s);
        flags = {z, c, s, o};
      end

      // MOVAM
      4'b0010 : begin
        out = operand1;
        flags = cflags;
      end
```

```
// MOVMA, MOVIA
4'b0011 : begin
  out = operand2;
  flags = cflags;
end




// AND, ANDI
4'b0100 : begin
  out = operand1 & operand2;
  z = (out == 8'b0000_0000 ? 1 : 0);
  flags = {z, cflags[2:0]};
end

// OR, ORI
4'b0101 : begin
  out = operand1 | operand2;
  z = (out == 8'b0000_0000 ? 1 : 0);
  flags = {z, cflags[2:0]};
end

// XOR, XORI
4'b0110 : begin
  out = operand1 ^ operand2;
  z = (out == 8'b0000_0000 ? 1 : 0);
  flags = {z, cflags[2:0]};
end

// SUBMA, SUBIA
4'b0111 : begin
  {c, out} = {1'b0, operand2} - {1'b0, operand1};
  z = (out == 8'b0000_0000 ? 1 : 0);
  s = out[7];
  o = (operand1[7] & ~operand2[7] & s) | (~operand1[7] & operand2[7] & ~s);
  flags = {z, c, s, o};
end






// INC
4'b1000 : begin
  {c, out} = operand2 + 1;
  z = (out == 8'b0000_0000 ? 1 : 0);
  s = out[7];
  o = (operand2 == 8'b0111_1111);
  flags = {z, c, s, o};
end

// DEC
4'b1001 : begin
  {c, out} = operand2 - 1;
  z = (out == 8'b0000_0000 ? 1 : 0);
  s = out[7];
  o = (operand2 == 8'b1000_0000);
  flags = {z, c, s, o};
end

// ROTATEL
```

```
4'b1010 : begin
  case (operand1[2:0])
    3'b000 : out = operand2;
    3'b001 : out = (operand2 << 1) | (operand2 >> 7);
    3'b010 : out = (operand2 << 2) | (operand2 >> 6);
    3'b011 : out = (operand2 << 3) | (operand2 >> 5);

    3'b100 : out = (operand2 << 4) | (operand2 >> 4);
    3'b101 : out = (operand2 << 5) | (operand2 >> 3);
    3'b110 : out = (operand2 << 6) | (operand2 >> 2);
    3'b111 : out = (operand2 << 7) | (operand2 >> 1);
    default : out = 8'bzzzz_zzzz;
  endcase
  flags = cflags;
end

// ROTATER
4'b1011 : begin
  case (operand1[2:0])
    3'b000 : out = operand2;
    3'b001 : out = (operand2 >> 1) | (operand2 << 7);
    3'b010 : out = (operand2 >> 2) | (operand2 << 6);
    3'b011 : out = (operand2 >> 3) | (operand2 << 5);

    3'b100 : out = (operand2 >> 4) | (operand2 << 4);
    3'b101 : out = (operand2 >> 5) | (operand2 << 3);
    3'b110 : out = (operand2 >> 6) | (operand2 << 2);
    3'b111 : out = (operand2 >> 7) | (operand2 << 1);
    default : out = 8'bzzzz_zzzz;
  endcase
  flags = cflags;
end




// SLL
4'b1100 : begin
  case (operand1[2:0])
    3'b000 : out = operand2;
    3'b001 : out = operand2 << 1;
    3'b010 : out = operand2 << 2;
    3'b011 : out = operand2 << 3;

    3'b100 : out = operand2 << 4;
    3'b101 : out = operand2 << 5;
    3'b110 : out = operand2 << 6;
    3'b111 : out = operand2 << 7;
    default : out = 8'bzzzz_zzzz;
  endcase

  if(operand1[2:0] == 3'b000) begin
    flags = cflags;
  end
  else begin
    z = (out == 8'b0000_0000 ? 1 : 0);
    c = operand2[8 - operand1[2:0]];
    flags =   {z, c, cflags[1:0]};
  end
end

// SRL
4'b1101 : begin
  case (operand1[2:0])
```

```
        3'b000 : out = operand2;
        3'b001 : out = operand2 >> 1;
        3'b010 : out = operand2 >> 2;
        3'b011 : out = operand2 >> 3;

        3'b100 : out = operand2 >> 4;
        3'b101 : out = operand2 >> 5;
        3'b110 : out = operand2 >> 6;
        3'b111 : out = operand2 >> 7;
        default : out = 8'bzzzz_zzzz;
    endcase

    if(operand1[2:0] == 3'b000) begin
        flags = cflags;
    end
    else begin
        z = (out == 8'b0000_0000 ? 1 : 0);
        c = operand2[operand1[2:0] - 1];
        flags =   {z, c, cflags[1:0]};
    end

end

// SRA
4'b1110 : begin

    // ">>>" does not work, which is very weird
    s = operand2[7];
    case (operand1[2:0])
        3'b000 : out = operand2;
        3'b001 : out = {{1{s}}, {7{1'b0}}} | (operand2 >> 1);
        3'b010 : out = {{2{s}}, {6{1'b0}}} | (operand2 >> 2);
        3'b011 : out = {{3{s}}, {5{1'b0}}} | (operand2 >> 3);

        3'b100 : out = {{4{s}}, {4{1'b0}}} | (operand2 >> 4);
        3'b101 : out = {{5{s}}, {3{1'b0}}} | (operand2 >> 5);
        3'b110 : out = {{6{s}}, {2{1'b0}}} | (operand2 >> 6);
        3'b111 : out = {{7{s}}, {1{1'b0}}} | (operand2 >> 7);
        default : out = 8'bzzzz_zzzz;
    endcase

    // CAUTION: reg s is reused here
    s = out[7];

    if(operand1[2:0] == 3'b000) begin
        flags = {cflags[3:2], s, cflags[0]};
    end
    else begin
        z = (out == 8'b0000_0000 ? 1 : 0);
        c = operand2[operand1[2:0] - 1];
        flags =   {z, c, s, cflags[0]};
    end
end

// COMP
4'b1111 : begin
    {c, out} = 0 - operand2;
    z = (out == 8'b0000_0000 ? 1 : 0);
    s = out[7];
    o = (operand2 == 8'b1000_0000);
    flags = {z, c, s, o};
end
```

```
        default : begin
          out = 8'bzzzz_zzzz;
          flags = 4'bzzzz;
        end

      endcase
    end


endmodule











module control_logic(

    input [1:0] state,
    input [11:0] ins,
    input [3:0] sr,

    output reg pc_e,
    output reg acc_e,
    output reg sr_e,
    output reg ir_e,
    output reg dr_e,

    output reg pmem_e,
    output reg dmem_e,
    output reg dmem_we,
    output reg alu_e,
    output reg [3:0] alu_mode,
    output reg mux1_sel,
    output reg mux2_sel,

    );

    reg [14:0] control = 15'b00000_0000_xxxx_xx;

    reg branch_taken = 0;


    always @(*) begin

      if(state == 2'b00) begin

        control <= 15'b00000_1000_xxxx_xx;
        {pc_e, acc_e, sr_e, ir_e, dr_e, pmem_e, dmem_e, dmem_we, alu_e, alu_mode, mux1_sel, mux2_sel} = control;

      end
      else begin

        casex(ins[11:8])
          4'b0000 : begin   // S Type, only NOP used
            case(state)
              2'b01 :   control = 15'b00010_1000_xxxx_xx;
              2'b10 :   control = 15'b00000_0000_xxxx_xx;
              2'b11 :   control = 15'b10000_0000_xxxx_1x;
            endcase
        end
        4'b0001 : begin // GOTO
          case(state)
            2'b01 :   control = 15'b00010_1000_xxxx_xx;
            2'b10 :   control = 15'b00000_0000_xxxx_xx;
            2'b11 :   control = 15'b10000_0000_xxxx_0x;
          endcase
        end
        4'b001x : begin // ALU M type
          case(state)
            2'b01 :   control = 15'b00010_1000_xxxx_xx;
            2'b10 :   control = 15'b00001_0100_xxxx_xx;
            2'b11 :   control = {1'b1, ins[8], 4'b1000, ~ins[8], ~ins[8], 1'b1, ins[7:4], 2'b11};
          endcase
        end
        4'b01xx : begin   // conditional branch
          case(state)
            2'b01 :   control = 15'b00010_1000_xxxx_xx;
            2'b10 :   control = 15'b00000_0000_xxxx_xx;
            2'b11 :   begin
              branch_taken = sr[ins[9:8]];
              control = {13'b10000_0000_xxxx, branch_taken, 1'bx};
              end
          endcase
        end
        4'b1xxx : begin // ALU I type
          case(state)
            2'b01 :   control = 15'b00010_1000_xxxx_xx;
            2'b10 :   control = 15'b00000_0000_xxxx_xx;
            2'b11 :   control = {9'b11100_0001, 1'b0, ins[10:8], 2'b10};
          endcase
        end

        default : control <= 15'b00000_0000_xxxx_xx;

      endcase

      {pc_e, acc_e, sr_e, ir_e, dr_e, pmem_e, dmem_e, dmem_we, alu_e, alu_mode, mux1_sel, mux2_sel} = control;

    end

  end

endmodule



module micro_controller();

  reg clk = 1'b0;

  reg [1:0] state = 2'b00;

  reg loading = 1'b1; // flag denoting loading program
  reg [7:0] pmem_la = 8'b0000_0000;
  reg [11:0] pmem_li = 12'b0000_0000_0000;
  reg pmem_le;


  // registers
  reg [7:0] pc; // = 8'b0000_0000;
  reg [11:0] ir; // = 12'b0000_0000_0000;
  reg [7:0] dr; // = 8'b0000_0000;
  reg [7:0] acc; // = 8'b0000_0000;
  reg [3:0] sr; // = 4'b0000;
  // wire connection
  wire [7:0] mux1_out;
  wire [7:0] adder_out;
  wire [11:0] pmem_i;
  wire [7:0] dmem_dout;
  wire [7:0] mux2_out;
  wire [7:0] alu_out;
  wire [3:0] alu_flags;


  // control signals
  wire pc_e;
  wire acc_e;
  wire sr_e;
  wire ir_e;
  wire dr_e;

  wire pmem_e;
  wire dmem_e;
  wire dmem_we;
  wire alu_e;
  wire [3:0] alu_mode;
  wire mux1_sel;
  wire mux2_sel;


  // components

  // FETCH group

  pmem pmem0(clk, pmem_e, pc, pmem_i, pmem_le, pmem_la, pmem_li);

  pc_adder adder0(pc, adder_out);

  mux_8b mux1(ir[7:0], adder_out, mux1_sel, mux1_out);


  // DECODE group

  dmem dmem0(clk, dmem_e, dmem_we, ir[3:0], alu_out, dmem_dout);

  // EXECUTE group

  mux_8b mux2(ir[7:0], dr, mux2_sel, mux2_out);

  alu_8b alu(acc, mux2_out, alu_e, alu_mode, sr, alu_out, alu_flags);

  // control logic

    control_logic cl(state, ir, sr, pc_e, acc_e, sr_e, ir_e, dr_e, pmem_e, dmem_e, dmem_we, alu_e, alu_mode, mux1_sel,
mux2_sel);


  initial begin

    clk = 1'b0;

    // load program, here the test program is hardcoded and loaded
    pmem_le = 1'b1;


    pmem_la = 0;
    pmem_li = 12'b0000_0000_0000; // NOP;
    #10
    pmem_la = 1;
    pmem_li = 12'b1011_0000_0001; // MOVIA   Acc, 1;            Acc = 1
    #10

    pmem_la = 2;
    pmem_li = 12'b1010_0000_0000; // RSV;                  (no operation)
    #10

    pmem_la = 3;
    pmem_li = 12'b1000_0000_0111; // ADDI   Acc, Acc, 7;      Acc = Acc + 7 = 1 + 7 = 8
    #10

    pmem_la = 4;
    pmem_li = 12'b1001_0000_0110; // SUBAI   Acc, Acc, 6;      Acc = Acc - 6 = 8 - 6 = 2
    #10

    pmem_la = 5;
    pmem_li = 12'b1111_0000_0111; // SUBIA   Acc, 7, Acc         Acc = 7 - Acc = 7 - 2 = 5
    #10

    pmem_la = 6;
    pmem_li = 12'b1100_0000_0011; // ANDI   Acc, Acc, 0x03;      Acc = Acc AND 0x03 = 0x05 AND 0x03 = 0x01
    #10

    pmem_la = 7;
    pmem_li = 12'b1101_0000_0101; // ORI   Acc, Acc, 0x05;      Acc = Acc OR 0x05 = 0x01 OR 0x05 = 0x05
    #10

    pmem_la = 8;
    pmem_li = 12'b1110_0000_0011; // XOR   Acc, Acc, 0x03;      Acc = Acc XOR 0x03 = 0x05 XOR 0x03 = 0x06
    #10


    pmem_la = 9;
    pmem_li = 12'b0001_0000_1001; // GOTO 9
    #10


    // Now loading is finished, clear loading flag

    pmem_le = 1'b0;
    loading = 1'b0;

    pmem_la = 8'b0000_0000;
    pmem_li = 12'b0000_0000_0000;


    #500
    $finish;

  end



  // generate clock for simulation
  always @(*)
    #5 clk <= ~clk;

  // implementation of the state machine
  always @(posedge clk) begin

    if(pc_e == 1'b1)
      pc <= mux1_out;
      if(ir_e == 1'b1)
        ir <= pmem_i;

      if(dr_e == 1'b1)
        dr <= dmem_dout;

      if(acc_e == 1'b1)
        acc <= alu_out;

      if(sr_e == 1'b1)
        sr <= alu_flags;


      case(state)
        2'b00 : begin
          if(loading == 1'b0) begin
            state <= 2'b01;

            pc = 8'b0000_0000;
            ir = 12'b0000_0000_0000;
            dr = 8'b0000_0000;
            acc = 8'b0000_0000;
            sr = 4'b0000;
          end
          else begin
            state <= 2'b00;
          end

        end

        2'b01 : state <= 2'b10;
        2'b10 : state <= 2'b11;
        2'b11 : state <= 2'b01;
        default : state <= 2'b00; // stay at load state
      endcase



    end


  endmodule
```