Examination

Design of Embedded DSP Processors, TSEA26

Date	8-12, 2017-10-19
Room	G34, G32, FOI hus G
Time	08-12AM
Course code	TSEA26
Exam code	TEN1
Course name	Design of Embedded DSP Processors Konstruktion av inbyggd DSP-processorer Written examination (skriftlig tentamen)
Department	ISY
Number of questions	5
Number of pages (including this page)	5
Course responsible	Dake Liu
Teacher visiting the exam room, phone	Dake Liu, 0702681256, 281256
Time to visit the exam room	About 09.00 and 11.00 (twice)
Course administrator	Gunnel Hässler
Permitted equipement	None, besides an English dictionary
Other important information *Grading*	Points Swedish grade 41-50 5 31-40 4 21-30 3 0-20 U
Number of exams in the bag	

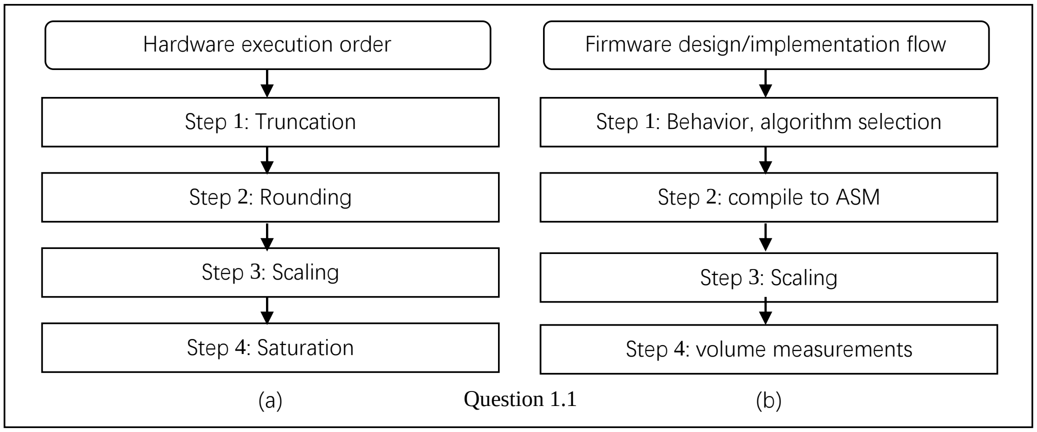
Important information:

- Please answer questions in English (if you cannot remember special technical words, you can use corresponding Swedish words).
- When designing a hardware unit you should attempt to minimize the amount of hardware. (Unless otherwise noted in the question.)
- The width of data buses and registers must be specified (with bit accurate annotations) unless otherwise noted. Likewise, the alignment must be specified in all bit accurate concatenations of signals or buses. When using a box such as "*SATURATE*" or "*ROUND*" in your schematic, you must (unless otherwise noted) describe the content of this box! (E.g. with RTL code).
- You can assume that all numbers are in two's complement representation unless otherwise noted in the question.
- In questions where you are supposed to write an assembler program based on pseudo code you are allowed to optimize the assembler program in various ways as long as the output of the assembler program is identical to the output from the pseudo code. You can also (unless otherwise noted in the question) assume that hazards will not occur due to parts of the processor that you are not designing.

Good luck!

Question 1: General questions (10p)

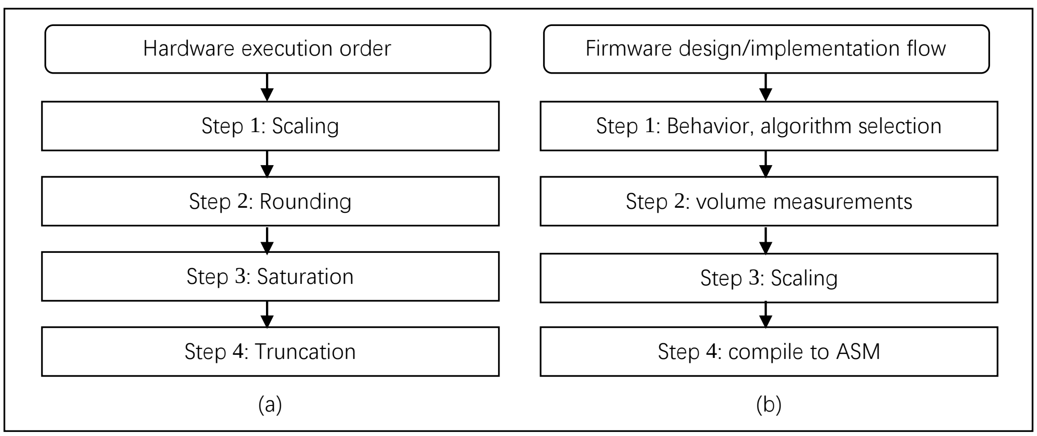
1.1. (2p) Are following two figures correct? If yes, state reasons, if wrong, give correct figures.



Question 1.1

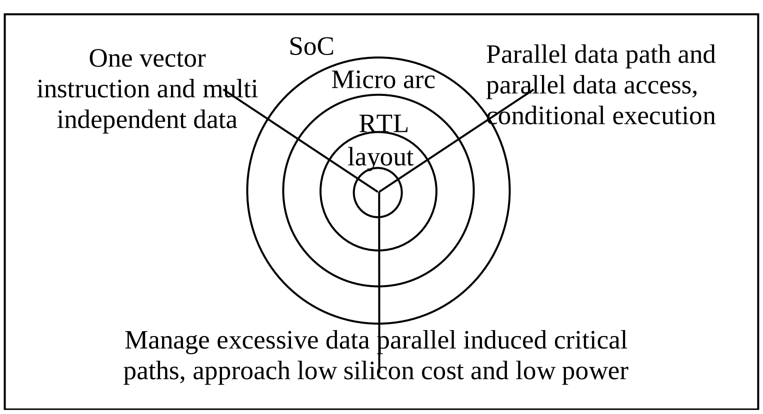
Solution

Tow figures are not correct. Correct figures are:



1.2. (1p) Using (Gajski) Y-chart and three sentences to describe a SIMD Processor

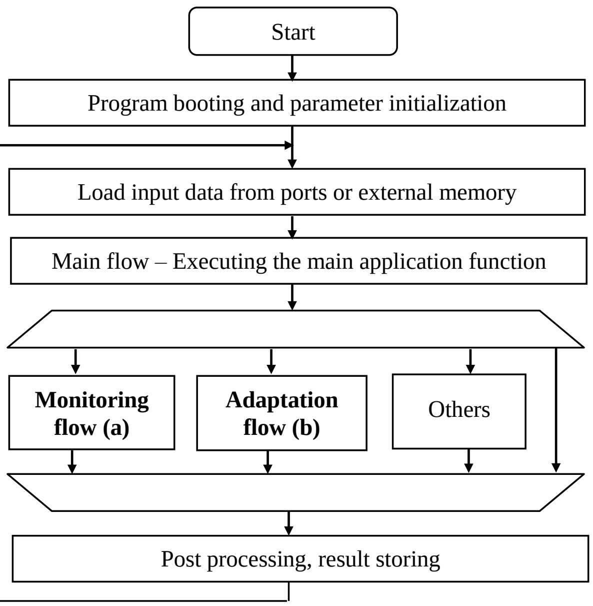
Solution



1.3. (1p) If data and twiddle factors are available in data memories, how many basic arithmetic operations and data access operations can be found while directly executing a DIT radix-2 butterfly algorithm for FFT using a RISC processor?

Solution, 10 arithmetic computing (4*, 6+) and 10 data access (6R, 4W)

1.4. (2p) Please describe:
1.4.1. What is the function in the monitoring flow (a) in the following flowchart? (0.5p)
1.4.2. What is the function in the adaptation flow (b) in the following flowchart? (0.5p)
1.4.3. Please find a DSP processor which does not require step (a) and (b) because of the special hardware and data type (1p).



Programming flow chart using fixed point ASIP

Solution
1, To measure the energy (volume) and the trends of it.
2, To adapt the scaling factors for the best dynamic range on results
3, floating point processor does not require step (a) and (b)

1.5. (1p) What shall a programmer prepare in master program for running a kernel task in a slave device (master and device are in two programming domains and share the same data set).

Solution
Prolog in mater include at least: loading codes to device, loading/preparing data for the task to run in device, configure the code accordingly, and finally, start the device task at the right time.

1.6. (1p) What is the difference between an arithmetic right shift and a logic right shift?

Solution: Arithmetic right shift filling sign bits after shift, logic right shift fill in zeros after shift

1.7. (2p) To implement an instruction set simulator ISS, you can execute an instruction by interpreting it or binary code translation. Please describe them and answer why a high quality ISS consists both interpreter and binary translator?

Solution: Interpreting means to emulate an assembly instruction of the target machine using a behavior function call. Binary translation is to find a host instruction the same as the target instruction, translate parameters, and directly run the host instruction. The reason is that we cannot find all host-target instruction pare for all target instructions.

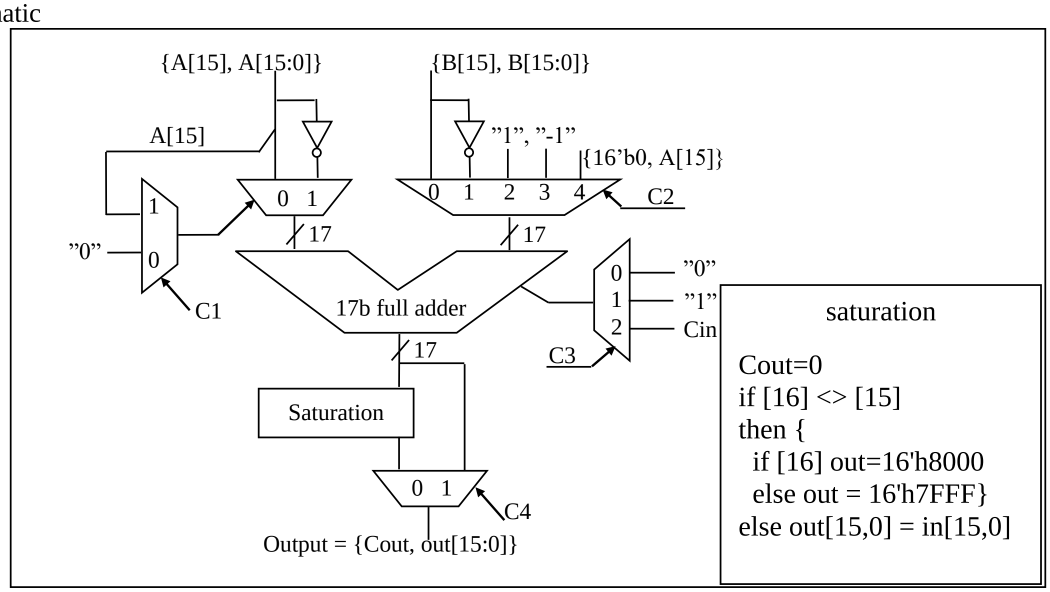
Question 2: ALU (8p)

3.1: (6p) Please design a 16b in/out arithmetic computing unit (AU) using only one adder and simple logic component such as multiplexer and logic gates. The AU is for a single step computing, it is not for iterative computing. Please design the circuit schematic drawing with complete connections and width annotations on each connection. The instruction subset of the arithmetic unit is given in the following table and *there are 10 instructions*. The operands A and B are from the general register file. Specify all control signals and finish a binary control table.

Instructions	**Function**
ADD with SAT	A = A + B with saturation
ADD without SAT	A = A + B without saturation
ADD with CIN SAT	A = A + B + Cin with saturation
ADD with CIN without SAT	A = A + B + Cin without saturation
SUB with SAT	A = A - B with saturation
SUB without SAT	A = A - B without saturation
CMP with saturation	SAT(A – B) set flags for compare
ABS(A)	A = ABS(A) Absolute operation, saturation
INC(A)	A = A+1 with saturation
DEC(A)	A = A-1 with saturation

Solution

Schematic

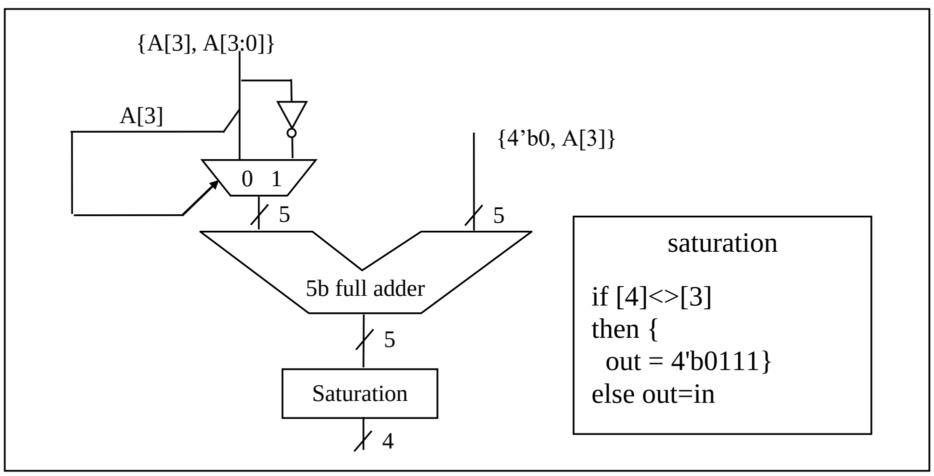


Control table

Instructions	**C1**	**C2**	**C3**	**C4**
ADD with SAT	0	0	0	0
ADD without SAT	0	0	0	1
ADD with CIN SAT	0	0	2	0
ADD with CIN without SAT	0	0	2	1
SUB with SAT	0	1	1	0
SUB without SAT	0	1	1	1
CMP	0	1	1	0
ABS(A)	1	4	0	0
INC(A)	0	2	0	0
DEC(A)	0	3	0	0

3.2. (2p) Design hardware for executing the ABS (A [3:0]) instruction, give bit accurate circuit schematic drawing, and try to completely verify the ABS function using three assembly instructions (do not need to verify the correctness of the full adder).

Solution:
Schematic



Verification
ABS(16’h8000) //to verify the corner
ABS(16’h7xxx) //to verify the ABS of a positive value.
ABS(16’h8xxx) //to verify the ABS of a negative value, xxx is not H000

Question 3: MAC (12p)

Implement instructions I1 to If, draw a schematic and design a control table. Instructions are listed:

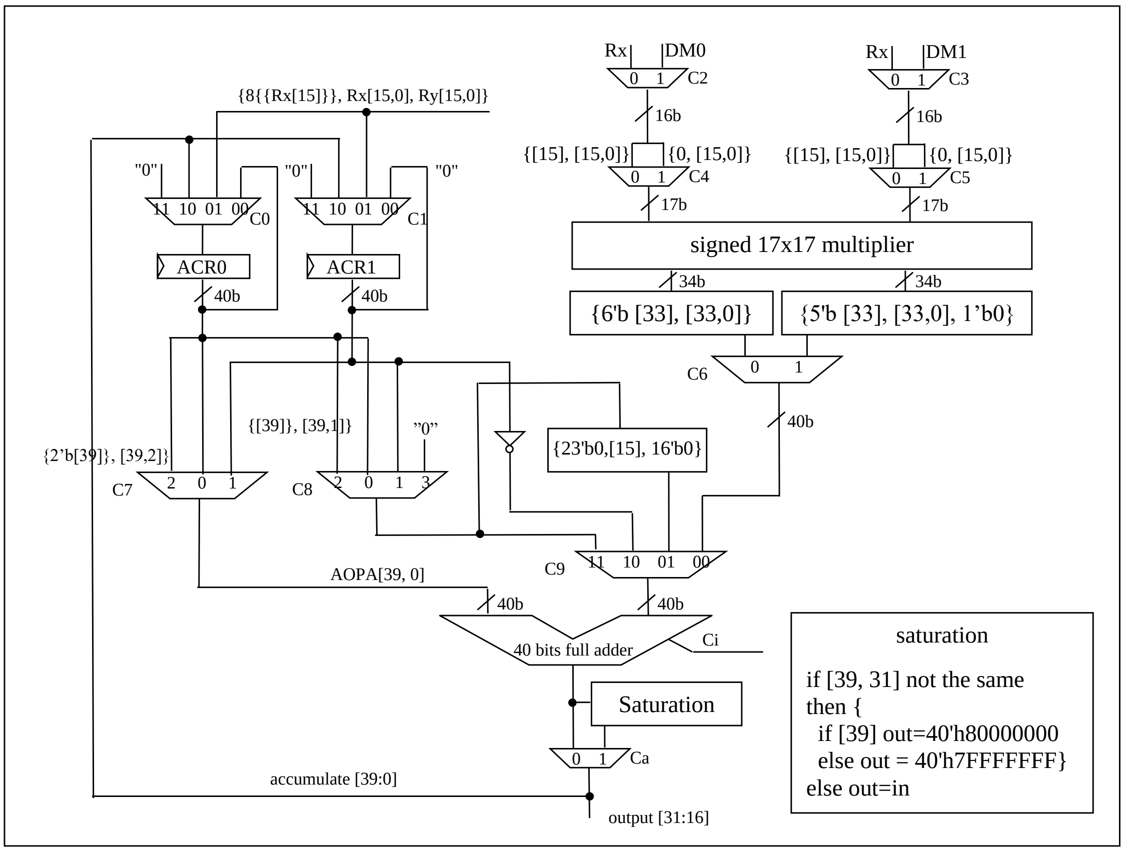
- `I1: NOP // No operation`
- `I2: ACR0 = 0`
- `I3: ACR1 = 0`
- `I4: ACR0 = {{8{RFx[15]}}, RFx[15:0], RFy[15:0]}`
- `I5: ACR1 = {{8{RFx[15]}}, RFx[15:0], RFy[15:0]}`

- `I6: ACR0 = ACR0 + RFx[15:0] * RFy[15:0] //signed integer multiplication`
- `I7: ACR0 = ACR0 + RFx[15:0] * RFy[15:0] //unsigned integer multiplication`
- `I8: ACR0 = ACR0 + RFx[15:0] * RFy[15:0] //signed fractional multiplication`
- `I9: ACR0 = ACR0 + DM0[ap0] * DM1[ap1] //signed fractional MUL`
- `Ia: ACR0 = ACR0 + ACR1`
- `Ib: ACR0 = ACR0 – ACR1`
- `Ic: ACR0 = Scaling(ACR0) //Scaling factor is 0.75`
- `Id: ACR1 = ACR0`
- `Ie: RFx[15:0] = SAT(ROUND(ACR0)) //move ACR0[31:16] to RFx`
- `If: RFy[15:0] = SAT(ROUND(ACR1)) //move ACR1[31:16] to RFx`

Constraints, inputs, outputs, and proposals:

- Both ACR0 and ACR1 are 40b accumulator registers and a RF is a 16-b general register,
- You shall use 17b x 17b signed multiplier as the primitive (component),
- You shall offer bit accurate annotations on connections,
- There are no *saturation* and *round* operations for instructions I1 to Id.

Solution



Control table

	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	Ca	Cin
I1	00	00	x	x	x	x	x	x	x	x	x	x
I2	11	00	x	x	x	x	x	x	x	x	x	x
I3	00	11	x	x	x	x	x	x	x	x	x	x
I4	01	00	x	x	x	x	x	x	x	x	x	x

I5	00	01	x	x	x	x	x	x	x	x	x	x
I6	10	00	0	0	0	0	0	0	x	00	0	0
I7	10	00	0	0	1	1	0	0	x	00	0	0
I8	10	00	0	0	0	0	1	0	x	00	0	0
I9	10	00	1	1	0	0	1	0	x	00	0	0
Ia	10	00	x	x	x	x	x	0	1	11	0	0
Ib	10	00	x	x	x	x	x	0	x	10	0	1
Ic	10	00	x	x	x	x	x	2	2	11	0	0
Id	00	10	x	x	x	x	x	0	3	11	0	0
Ie	10	00	x	x	x	x	x	0	0	01	1	0
If	00	10	x	x	x	x	x	1	1	01	1	0

Question 4: Address Generation Unit (AGU) (10p)

An AGU supports fast access of matrices. The AGU should support the incremental features to generate the index of the next matrix element. The AGU shall support matrices size ($M \times N$). The matrix is stored in memory in row-major order. All matrix elements can thus be accessed by the following addresses relative to the first matrix element (element 0):

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{pmatrix}$$

To start the matrix data access, an address register always points to the first element of the matrix (element 0), after which the AGU should generate either the next element in the same row (row-major order), or the next element in the same column (column-major order). While reaching the last element of a row(/column), the AGU should wrap around and continue with the next row(/column).

Assuming that the address register is initially set to 0, row-wise increment should generate the following address sequence (for the example matrix):

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

Column-wise addressing should instead generate the following sequence:

```
0 4 8 12 1 5 9 13 2 6 10 14 3 7 11 15
```

To transpose a matrix, see the assembly example below. The syntax *ar0:+=next(row)* indicates increment to the next row element, and *ar1:+=next(col)* the next column element. A *nop* instruction is inserted between the load instruction and the store instruction to avoid access conflict.

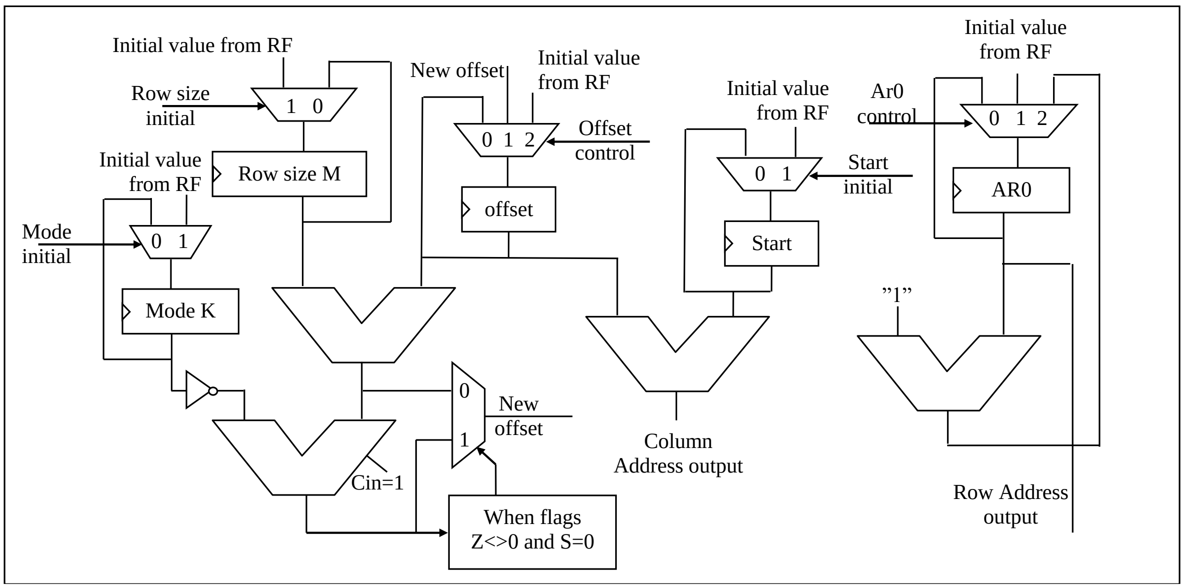
```
transpose:
   // some setup code (which configures the AGU)
   // ...
   repeat $(M*N) loop_end // repeat M*N times
      load r0,DM0[ar0:+=next(row)]
      nop
      store DM0[ar1:+=next(col)],r0
   loop_end:
   ret <ds0>
```

(a) (8p) Draw a schematic and control table of your AGU.

(b) (2p) Demonstrate the use of your AGU, by completing/modifying the pseudo assembly code above. Special-purpose register(s), including M and N, can be loaded through general register. Address registers *ar0* and *ar1* have been initialized to the starting value of the input and output matrix respectively.

Solution:

1. Initial: Start = The first element address of the matrix;
2. Initial offset = 0; Row size = N; Column size = N; Mode K = M*N-1
3. The first (row) addressing model is A=Start + post increment offset
4. The second (column) addressing model is A=Start + (offset + row size) mod K
5. The schematic



6. The control table

	Mode initial	Start initial	Row size initial	Offset control	Ar0 control
Load row size M	0	0	1	0	0
Load start	0	1	0	0	0
Load Ar0	0	0	0	0	1
Load mode K	1	0	0	0	0
Offset initial	0	0	0	2	0
Row mode	0	0	0	0	2
Column mode	0	0	0	1	0

7. The assembly code

Specify the row addressing mode as AR `ar0:+=next(row)`

Specify the column addressing mode as AC `ar0:+=next(row)`

```
Move M R15     // suppose that M is in R15
Move Start R14 // suppose that Start is in R14
Move Ar0 R14
Move K R13     // suppose that K is in R13
repeat 16 loop_end // repeat M*N times
      Load r0,DM0[AR]
      Nop
      Store DM0[AC],r0
      Nop
loop_end:
```

Question 5: Program flow control (10p)

The pipeline specification is:

1. P1: Pointed an instruction
2. P2: Fetched an instruction and stored it into the instruction register
3. P3: An instruction is decoded and decoded signals are latched
4. P4: Data from register file are ready on In-ports of the multiplier in a MAC
5. P5: Data are ready on accumulator inputs in a MAC
6. P6: MAC Flags are ready to use

Design part of the control path: The design shall include functions:

4.1. PC[15:0] <= 0; Reset, and starts executing at address 0x0000 after reset,
4.2. PC[15:0] <= PC+1 as the default of the PC FSM,
4.3. PC[15:0] <= immediate [15:0]; unconditional jump, immediate is carried by the jump instruction. The decoding of the unconditional jump shall be used for jump decision before latching (pipelining),
4.4. PC[15:0] <= RF[15:0] when conditional jump is taken, RF here is a register value from the general register file, The condition is from MAC flags.

The design outputs shall include

1. Design pipeline execution tables for unconditional and conditional jumps,
2. Draw pipeline accurate schematic circuits, and implement functions 4.1, 4.2, 4.3, and 4.4,
3. Design control signal for controlling the Next PC.

Solution

In following tables, MAC is a MAC instruction, JMP is an unconditional jump, MCJMP is MAC conditional jump instruction, FI is a following instruction

Pipeline execution table for unconditional jump

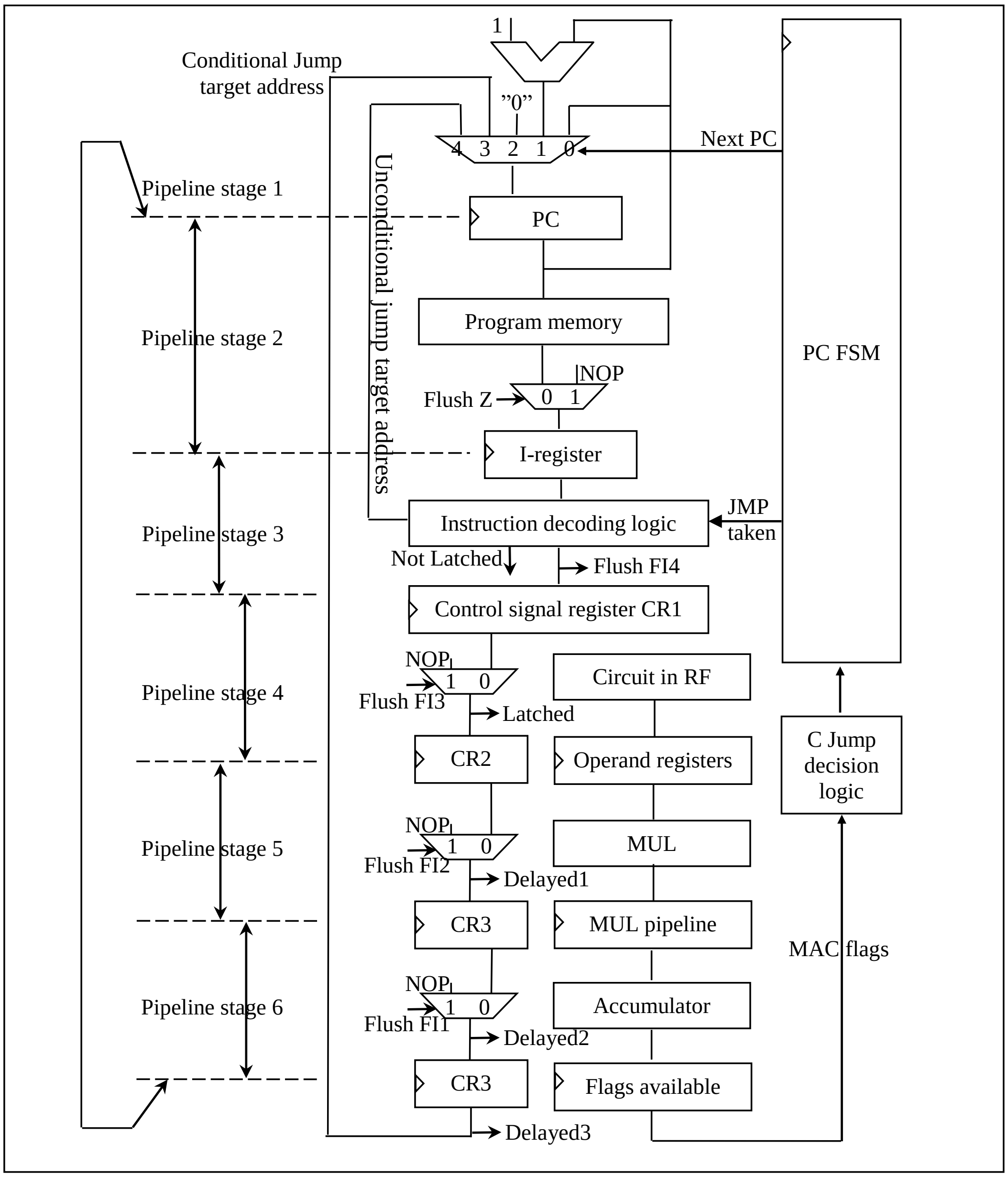
clock	P stage 1 Instruction pointed	P stage 2 Instruction fetched	P stage3 Instruction decoded	P stage 4 Operand fetched	P stage 5 ALU/MUL
1	JMP to B	---	---	---	---
2	FI	JMP	---	---	---
3	B: MUL R1 R2	Flush FI	JMP	---	---
4	---	B: MUL R1 R2	NOP	JMP	---
5	---	---	B: MUL R1 R2	NOP	JMP
6	---	---	---	B: MUL R1 R2	NOP
7	---	---	---	---	B: MUL R1 R2

Pipeline execution table for conditional jump

clock	P stage 1 Instruction pointed	P stage 2 Instruction fetched	P stage3 Instruction decoded	P stage 4 Operand fetched	P stage 5 MUL	P stage 6 MAC flag available
1	MAC	---	---	---	---	---
2	MCJMP B	MAC	---	---	---	---
3	FI 1	MCJMP B	MAC	---	---	---
4	FI 2	FI 1	MCJMP B	MAC	---	---
5	FI 3	FI 2	FI 1	MCJMP B	MAC	---
6	FI 4	FI 3	FI 2	FI 1	MCJMP B	MAC
7	B: ADD R1 R2	Flush F4	Flush F3	Flush F2	Flush F1	MCJMP B

8		ADD R1 R2	FI4 =NOP	FI3 =NOP	FI2 =NOP	FI1 =NOP
9			ADD R1 R2	FI4 =NOP	FI3 =NOP	FI2 =NOP
10				ADD R1 R2	FI4 =NOP	FI3 =NOP
11					ADD R1 R2	FI4 =NOP
12						ADD R1 R2

Schematic



Control signal for the Next PC

```
If reset=0 then Next_PC=2
Elseif MCJMP && Delay3 && MAC_flags_true then Next_PC=4
Elseif JMP_taken then Next_PC=3
Elseif single_instruction_loop then Next_PC=0
Else Next_PC=PC+1
```