14 Control Path Design

Hardware organization and micro architecture implementation of control path will be discussed in this chapter. Micro architecture implementation of PC FSM, and instruction decoder will be discussed in detail.

14.1 Control Path in General

14.1.1 General

A control path is a hardware module in a processor core. It handles both synchronous tasks (running program codes) and asynchronous tasks (handling exceptions). Two main parts of hardware can be found in a control path: the instruction flow controller and the instruction decoder.

The control path in a basic ASIP or a DSP processor is illustrated in the left part in Figure 14-1 with gray background. Inputs of a control path include loaded instruction code from PM (program memory), flags from datapath, branch target address, configuration vectors, and asynchronous control requests. Outputs of a control path include control signals to datapath, to address generator, to control path itself, as well the immediate data from the decoded instruction.

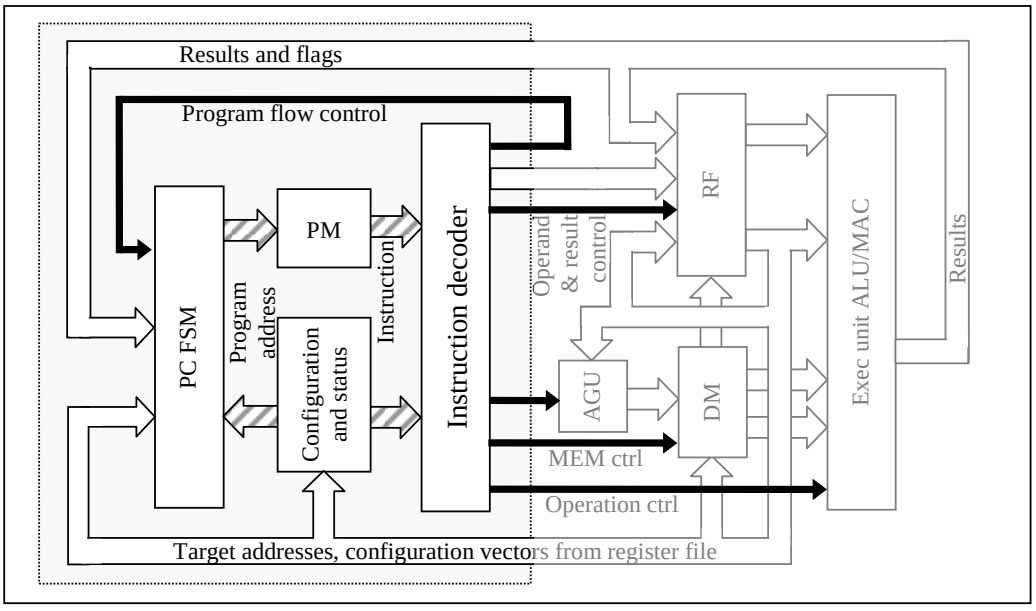


Figure 14-1 Control path in a DSP processor

The PM address is generated from the PC finite state machine (FSM). An instruction is fetched by the PC FSM according to the control of the program flow.

The fetched instruction is decoded and control signals are generated. Some of the

control signals go to the address generator for address generation and memory controls. Some of the control signals go to the datapath, controlling register file access and data manipulation. Some of the control signals stay in control path for the decision of the next instruction and the decision of the instruction to decode (fetched or NOP). In most cases, the instruction decoder does not supply control signals to peripherals. Peripherals have their own control FSM. Its FSM can be controlled by processor core via reading and writing peripheral registers (special registers). A typical and simplified control path is depicted in the following Figure 14-2.

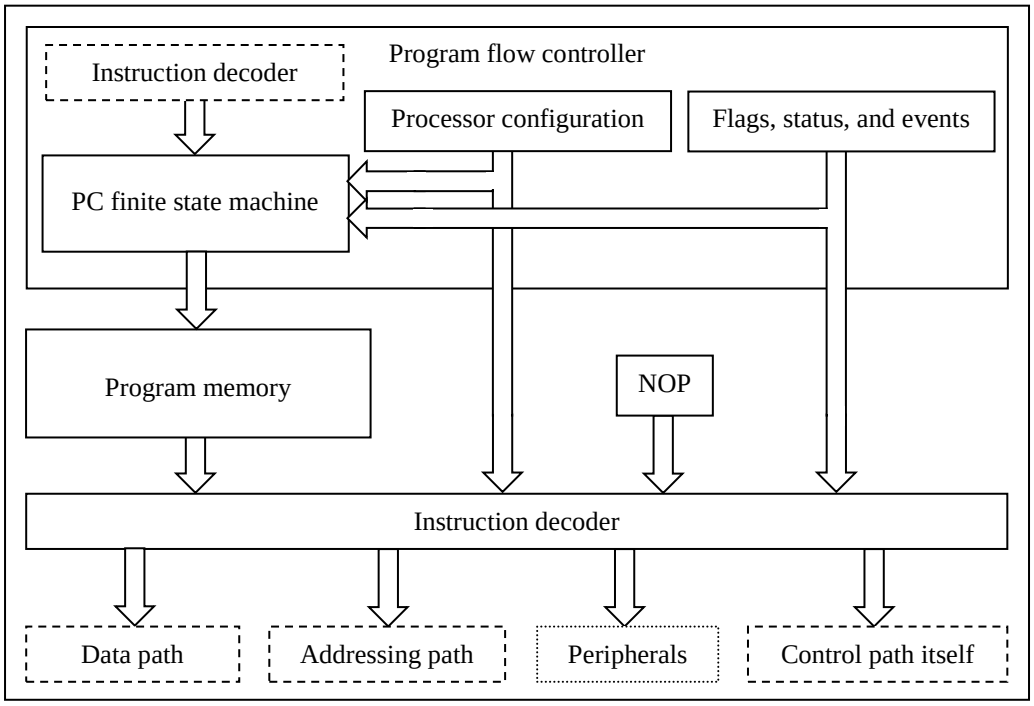


Figure 14-2 Simplified function description of control path

14.1.2 Executing an Instruction

Before going into the details of the control path, readers need to know "how an instruction is executed in a processor". Most ASIP/DSP processor architectures are based on Harvard architecture. In Figure 14-3, the instruction execution procedure is illustrated by a flow chart.

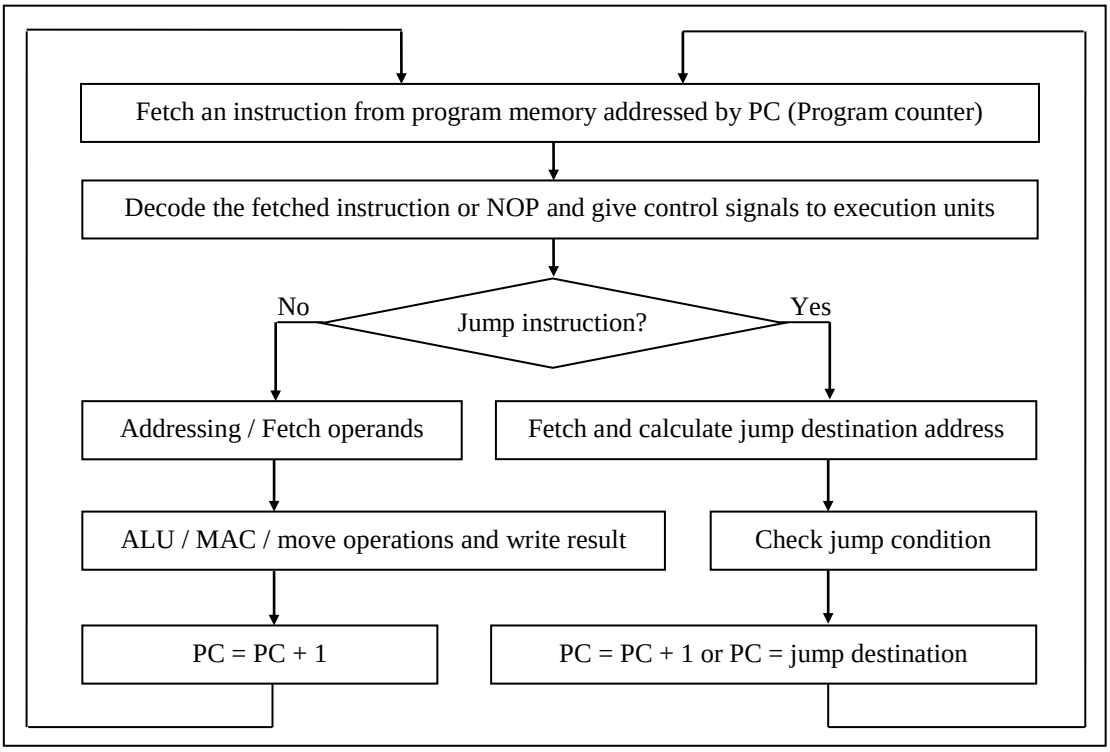


Figure 14-3 Procedure of executing an instruction

In this flow chart, an instruction (or a NOP) is fetched from program memory. After being decoded by the instruction decoder, the instruction will be executed either in the path of "data computing or moving" or in the path of "branch execution". Here a branch instruction could be a normal jump or a function call and return. In Figure 14-3, pipelining and instruction level parallelization have not been mentioned, these will be discussed in detail later in this chapter.

14.2 Control Path Organization

The design of a control path consists of two steps: the control path organization and the micro architecture hardware implementation. The organization includes functional specification, partition, allocation and scheduling of pipeline.

14.2.1 Pipeline

14.2.1.1 The Processor Pipeline

Pipeline concept was discussed in chapter 3. It will be discussed in this chapter in detail. To partition jobs into several pipeline stages, the system speed can be promoted much because the speed of each stage can be much faster and all pipeline stages are running in parallel.

By dividing parallel execution steps into n independent hardware pipeline staps,

speed up can be ideally up to n times. However, the task partition cannot be balanced into each pipeline, the pipeline speed is limited by the slowest pipe-step; and clock uncertainty induced by physical clock skew and clock jitter will both limit the speed. The processor performance enhanced by the pipeline architecture is measured using the following formula:

$$Speedup = \frac{n}{1 + stall_{cycles} / total_{cycles}} \times \frac{Period_{clock} - skew_{clock}}{Period_{clock}} \qquad \textbf{Equation 14-1}$$

Here n is the pipeline depth. The pipeline stall cycles ($stall_{cycles}$) can be calculated based on statistics. For example, by running one application which consumes $total_{cycles}$=2000 clock cycles, the number of jump-taken is 20 and the stall caused by every jump-taken takes 3 cycles. The cycle overhead caused by pipeline stalls is $stall_{cycles}/total_{cycles}$= 20*3/2000=3%. The clock skew or clock uncertainty is the deviation of clock arrival time V.S. the ideal arrival time. The deviation increases when design scale (chip size) is larger and silicon feature size is smaller. When clock rate is higher, the impact of clock uncertainty is relatively larger. When the clock rate is ultra-high (>2GHz for example), the clock uncertainty becomes dominant and to increase the pipeline depth may actually decrease the system speedup.

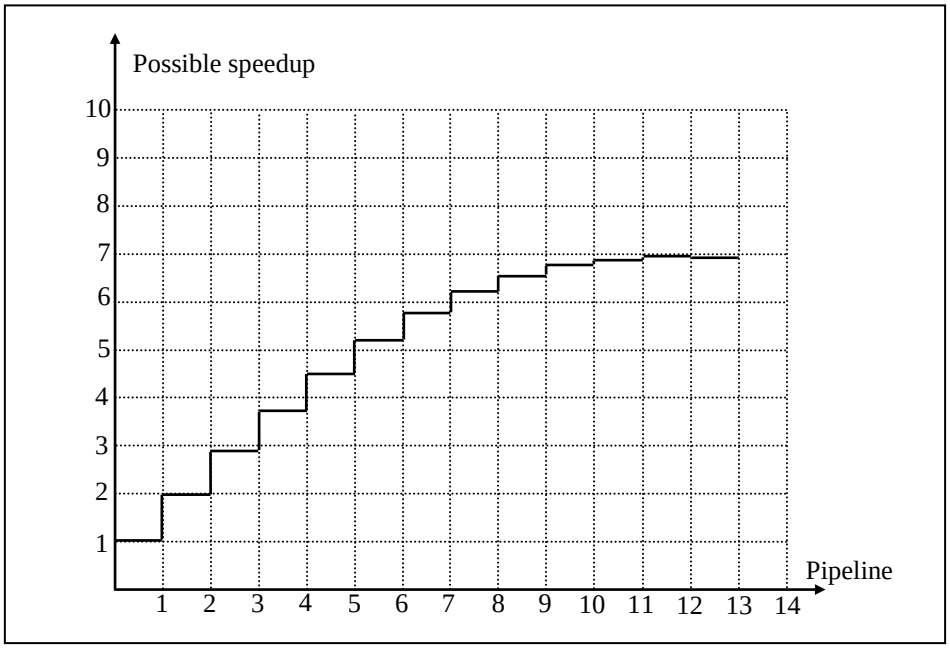


Figure 14-4 Speedup versus the number of pipeline stages

When logic path delays in pipeline steps are relative balanced, the best pipeline induced speed up can be estimated based on statistics of designs and parameters from semiconductor foundries. The plot in Figure 14-4 is based on the assumption that the critical paths in each pipeline stage are equal. Real designs are much different, because the critical path cannot be equal in each pipeline step; the real speedup is lower than that in Figure 14-4. Experiences show that the pipeline depth

for low cost and low power simple ASIP DSP can be around 3 to 7.

14.2.1.2 Definitions of Processor Pipelines

In the following text, typical pipeline stages in an ASIP DSP are briefly described on micro architecture level.

Pipeline stage 1: Instruction Fetch (IF): In this pipeline stage, the operation is to read a new instruction from the program memory using the PC value as the memory address. The operation is formalized in the following formula and the functional block diagram is illustrated in Figure 14-5.

```
instruction_register <= program_memory[program_counter];
```

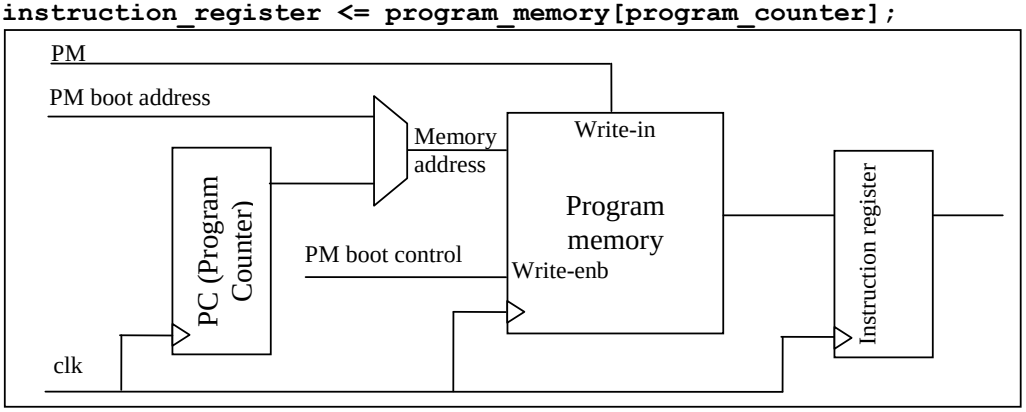


Figure 14-5 Instruction fetch hardware and pipeline

Pipeline stage 2: Instruction Decoding (ID): The fetched instruction is stored in the instruction register. In this clock cycle the fetched instruction (or NOP) will be decoded. Decoded control signals might be clocked or not (immediately used). The function of the instruction decoder is described by the following formula:

```
control_signal
  <= decoding_logic (instruction/NOP, processor_configuration, status);
```

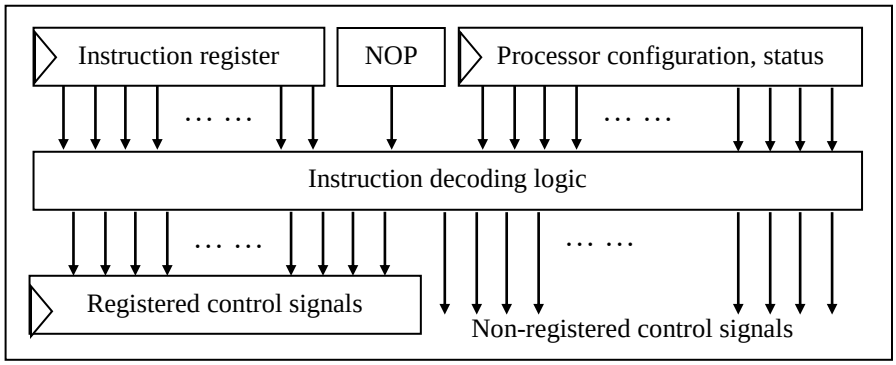


Figure 14-6 Instruction decoding

Pipeline stage 3: Operand fetch (OP): Operands can be fetched either from the register file or from the data memories. The memory address or register address should be provided during the instruction decoding. During the OP pipeline stage,

the operand is supplied to the input of the computing devices in the datapath. The operation of OP is given in the following formula:

```
operand <= operand_source (operand address);
```

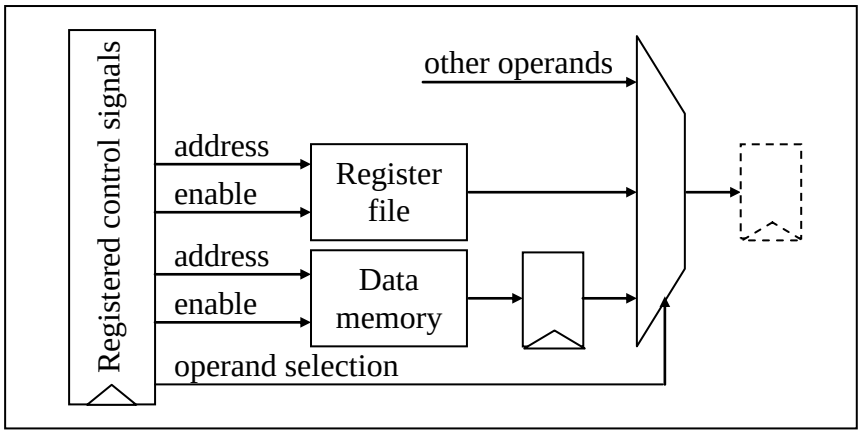


Figure 14-7 Operand fetch

Pipeline stage 4: Execution (EX): The execution of an instruction may take one clock cycle or multiple clock cycles. The ALU, register file, or memory access operation normally takes one clock cycle, while the MAC operation usually takes two or more cycles. Normally, the execution of the MAC instruction in the first clock cycle is multiplication and the execution in the second clock cycle is accumulation.

```
result <= OP (operand A; operand B; control_signals);
```

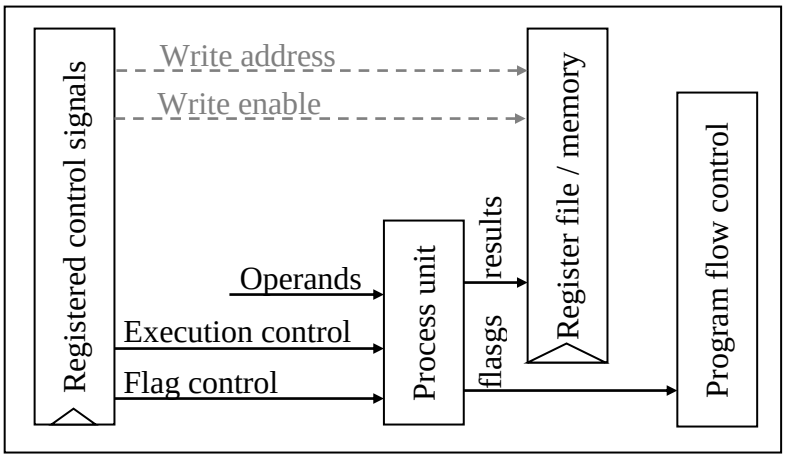


Figure 14-8 Pipeline stage for execution

Pipeline stage 5: Write back or store result (WB): Storing result may take one clock cycle or in EX step. In some processors, when the write (input) port of the register file is complicated, one cycle is necessary for writing back the result. The function is described by:

```
Register_file [address] <= result;
```

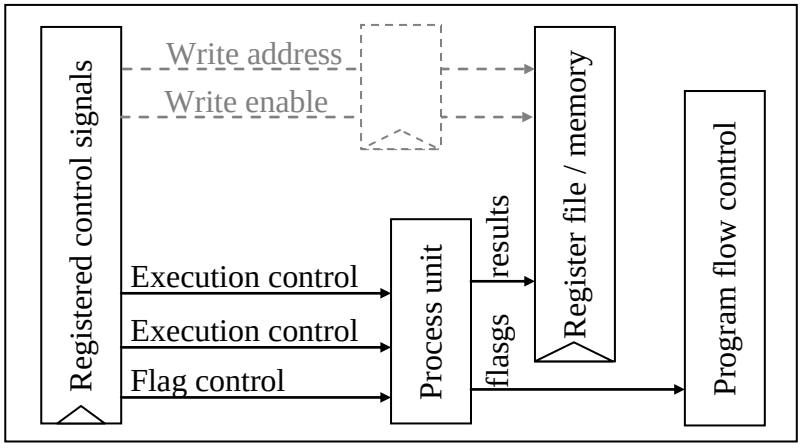


Figure 14-9 Pipeline stage for result store

Example 14-1 Design pipeline architecture for "Senior" processor according to the specification given in the Appendix of the book.

The Appendix gave a simplified pipeline structure. Refinement of the pipeline is:

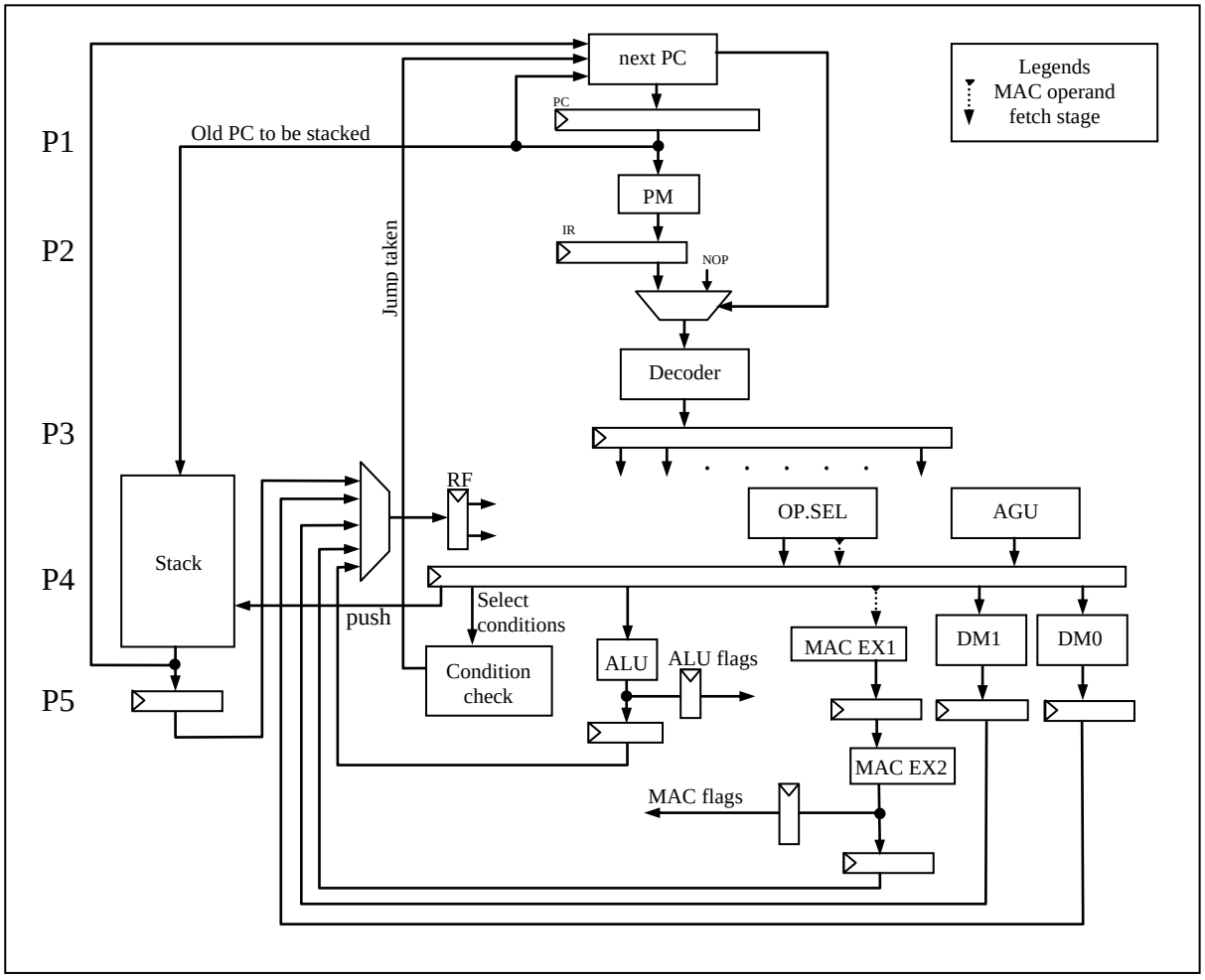


Figure 14-10 The pipeline design of the Senior DSP core

The pipeline design is based on 4/6-stages pipeline architecture it is:

1. Instruction fetch
2. Instruction decoding
3. Operand Fetch and Memory address computing
4. Execution cycle one: ALU, MUL (for MAC), Register access, Memory access
5. Execution cycle two: Accumulation, Multiplication for convolution (CISC)
6. Execution cycle three: Accumulation for convolution (CISC)

The pipeline for convolution (not for MAC) is special because the operand is from the output of data memories.

Analysis: The pipeline structure is simple and efficient. There might be a hidden critical path: That is the added delay to the pipeline 5, the multiplication for the convolution. The added delay is from the (possible long) interconnection between the memory output port to the multiplier in port.

14.2.2 Hazards and hazard handling

Hazards were discussed in previous Chapters. In this chapter, we review hazards and introduce the hazard design. Processors are based on the pipelined parallel principle. The pipeline concept introduction on chapter 14.2.1 was illustrative and did not cover hazard. Actually, the central challenge in pipelined systems is to handle hazards. Hazards happen when the next instruction cannot be executed in the next clock cycle, or the execution leads to incorrect results.

Three common types of hazards are data hazards, structural hazards, and control flow hazards (branching hazards). In following sub-sections, three hazards will be discussed as the processor design fundamental knowledge. Hazard handlings will also be discussed through design of PCFSM (Program Counter Finite State Machine), design of compiler, and programming.

Though the best way of hazard handling is to design a superscalar with OOO (Out Of Order) Execution, however, it is out of the scope of the book. In this book, hazards handling is based on the static way (time-stationary) instead of the dynamic way (superscalar or data stationary).

14.2.2.1 Data Hazards

Data hazard was mentioned several times in this book, we summarized the concept here. A data hazard occurs when instructions with data dependence modifying data in different stages of a pipeline. Considering that there are two instructions, instruction1 and instruction2, with instruction1 occurring before instruction2 in following program order. There are three situations data hazard may occur:

Case one, read after write (RAW):

It is a true dependency. The second instruction tries to read an operand (register) before the first instruction writing to it. That is, the needed data is not yet available

while fetching/using. A read after write (RAW) data hazard refers to a situation where an instruction refers to a result that has not yet been calculated or retrieved. This usually occur because even though an instruction is executed after a prior instruction, the prior instruction has been processed only partly through the pipeline, not yet finished, or not yet written back.

Example 14-2 RAW data hazard

```
instruction1. R2 <- R1 + R3
instruction2. R4 <- R2 + R3
```

The first instruction is calculating a value to be saved in register R2, and the second instruction is going to use this value to compute a result for register R4. However, when operands are fetched for running the 2nd instruction, the results from the first is not yet saved. (Result store is at the next cycle after the execution cycle.) Data in R2 is still the old value, the running of instruction 2 will be wrong. Hence a data dependency occurs.

Conclusion: A data dependency occurs while running instruction 2, as it is dependent on the completion of instruction 1.

Case two, write after read (WAR):

It is an anti-dependency. In this case, the instruction 2 tries to write a result to the destination register R5 before the R5 is read by instruction 1. A write after read (WAR) data hazard represents a problem with concurrent execution.

Example 14-3 WAR data hazard

```
instruction1. R4 <- R1 + R5 | instruction2. R5 <- R1 + R2
```

Whenever instruction2 may finish before instruction1 with concurrent execution, the processor design must be ensured that the result of register R5 is not stored before instruction1 has a chance to fetch the operands.

Case three, write after write (WAW):

It is an output dependency. In this case, the instruction2 tries to write result before it is written by instruction1 (though it might be a coding bug). A write after write (WAW) data hazard may occur in a concurrent execution environment.

Example 14-4 WAW data hazard

```
instruction1. R2 <- R4 + R7 | instruction2. R2 <- R1 + R3
```

The write back (WB) of instruction2 must be delayed until instruction1 finishes its execution. To conclude, WAR and WAW could be easily handled in ASIP because the architecture uses to be simple for only a domain of applications. RAW is the focus of the data hazard handling.

The easiest way to handle data hazard is to design an instruction scheduler in a

compiler. The principle is: 1) to check data dependency, 2) to reorder executions of independent instructions, and 3) to insert NOP when there is no independent instruction available.

Data hazard can also be handled during assembly coding. A typical example is to avoid data hazard after executing a multi-cycle instruction. For example, to modify the assembly codes in the following example.

Example 14-5 Assembly coding to avoid data hazard

```
ACR1 <- ACR1 + R1 * R2 // It takes 2 clock cycles
NOP // wait one clock cycle
R3 <- Saturate (round(ACR1)) // get the right result from ACR1
```

In most of cases, data hazards shall be handled during the hardware architecture design. For example, register forwarding technique……

14.2.2.2 Register (operand) forwarding

Data hazards can also be handled through register forwarding. Let us check the typical example:

Example 14-6 Run following assembly instructions using the datapath in Figure 14-11 (a)

```
Initial values in the register file: R1 = 1 and R2 = 2
Instruction 1: R1 = R1 + R2
Instruction 2: R4 = R1 + R2 //should be R4 = (1+2)+2 = 5
```

However, after the execution, we actually find the result is R2=3 instead of R2=5. Let us analyze the processor structure in Figure 14-11 and find out the reason.

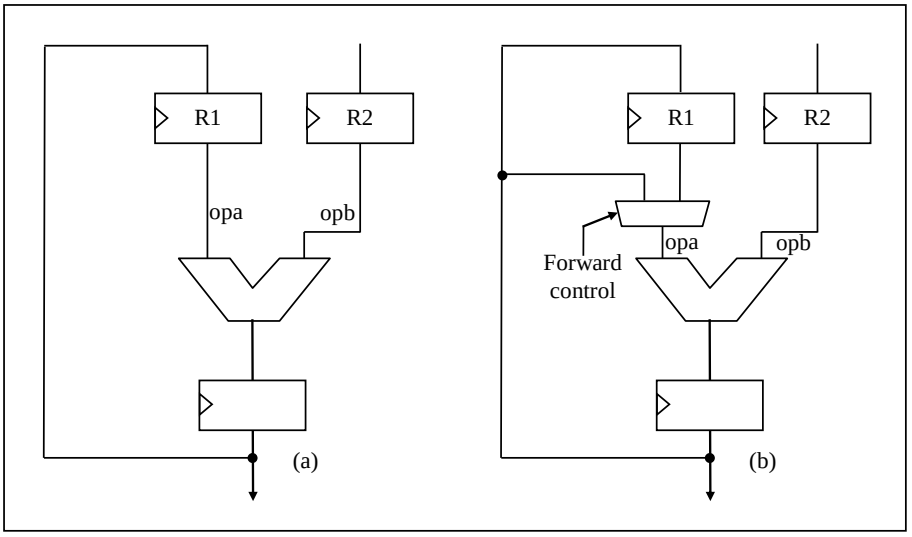


Figure 14-11 Register (operand) forwarding

We first look into the left figure (a). When we run an instruction, the data already available in a register of the register file is used as an operand, the result is buffered in the result buffer register; the result will be stored in the register file at the next

cycle. It means that the stored result cannot be used as an operand during the next execution cycle, see the following table.

Table 14-1 Register values in Figure 14-11 (a)

Clock cycles	Operation		R1Q	R2Q	opa	opb
1	Buffer<=R1+R2		1	2	1	2
2	R4<=R1+R2	R1<=Buffer	1	2	1	2
3		R4<=Buffer	3	2		
4	Result in R4=3 is wrong					

The table expose a typical data hazard, the needed operand following coding the order cannot be available as the programmer expected.

Register forwarding is designed in Figure 14-11 (b). In (b), a result of an instruction will be written to the register file at the write back cycle. At the same time, the result can be forwarded as an operand through pipeline by "forwarding control". The generation of forwarding control signal will be discussed during the discussion of instruction decoding in this chapter. The correct result is in table 14-2 with register forwarding in Figure 14-11 (b).

Table 14-2 Register values in Figure 14-11 (a)

Clock cycles	Operation		R1Q	R2Q	opa	opb
1	Buffer<=R1+R2	opa<=R1+R2	1	2	1	2
2	R4<=R1+R2	R1<=Buffer	1	2	3	2
3		R4<=Buffer	3	2		
4	Result in R4=5 is correct					

14.2.2.3 Branching Hazards

A branching hazard is also called control flow hazard. It occurs with branches. In any instruction pipeline microarchitectures, there are two cases:

1. The processor actually does not know if the currently fetched instruction is a branch instruction or not, because the fetched instruction is not yet decoded. Thus, the processor will keep fetching the next instruction. However, the fetch of the instruction right after a unconditional branch is definitely a wrong fetch.
2. If a fetched instruction is a conditional branch, several following instructions might be fetched correctly (if the branch is not taken) or several following fetched instructions will be fetched wrong (if the branch is taken). The time to know the right/wrong fetch will be when the branch condition is executed.

When instruction fetches are not correct, the fetched instructions must be flushed (clean all instructions to be executed in hardware in each pipeline step) and executions / writing back results will absolutely forbidden, because the results are not expected by the programmer.

Branch hazard must be avoided in the control path design. There are ways to handle

branch hazards. 1). Insert NOP instructions during compiling/programming; 2). Insert instructions independent to the branch taken/not taken; 3). Hardware stall (it is an operation called pipeline bubbling, a stall waits for a branch decision); and finally, 4). Design for flushing. Flushing or waiting for branch decision will cost extra clock cycles, weaken the processor performance. Branching hazard handling design will be discussed after the PC FSM introduction.

14.2.2.4 Structural Hazards

A structural hazard is actually a hardware design bug. A structural hazard means a structural design bug in time-stationary architecture (simple scalar) or a dynamic resource handling bug in data-stationary architecture (superscalar). It happens when a part of the processor's hardware is needed by two or more instructions at the same time. For example, a single memory unit is accessed in the fetch stage to get data, and accessed by previous instruction to write a data.

They can often be resolved by separating the component into orthogonal units (such as separate caches) or bubbling the pipeline.

14.2.3 Instruction Flow Controller

An instruction flow controller (program flow controller) updates the program counter (PC) for fetching the next instruction. A basic instruction flow controller is a finite state machine (PC FSM) with the state diagram illustrated in Figure 14-12.

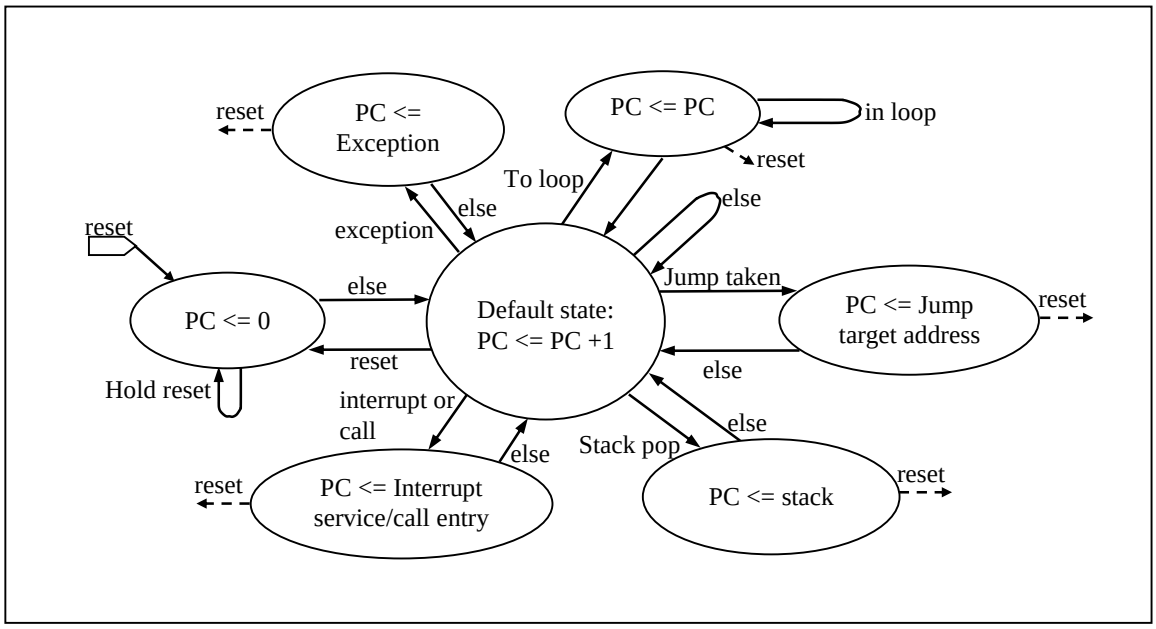


Figure 14-12 Example of a PC FSM

The PC FSM points to the next PC with the address of the next instruction to be fetched in the program memory. The default state is the PC<=PC+1. The initial state of the PC FSM is the reset and holding of the reset. Reset may happen at any time (see

the dash lines). One special state is PC<=PC, which is used to support single instruction hardware loop. While running a hardware loop instruction, for example the convolution instruction, the PC FSM holds the current instruction for N cycles. Another special state is the stack pop, which is the return function when finishing an interrupt/call. The rest of the states are used for supporting different kinds of jumps or other program flow control functions. The PC FSM can also be specified using a state transfer table. Conditions for the next state in the table are depicted as follows:

Table 14-3 Decision of the next PC

Next PC	Decision priority	Conditions
PC <= 0	Highest	Reset and hold on reset
PC<= exception handling or interrupt service	2nd	Exception or Accepted interrupt entry
PC <= jump target address	3rd	CALL or Jump taken and jump on a constant
PC <= Stack pop	3rd	Return from a CALL or an interrupt
PC <= PC	3rd	To a loop and In a loop
PC <= PC + 1	Lowest	Default

Conditions of a conditional jump are flag values from the results of the latest previously executed arithmetic instructions. "Default" in Figure 14-12 means "if there is no other condition". The condition "In a loop" means a simple loop is executed and the loop finish flag is not yet true. The logic design should follow the decision order as depicted in Table 14-3. "First" means the highest decision priority and "Lowest" means the lowest decision priority. Jump taken decisions are from the computing of the current flags (for example, in Table 14- 4).

Table 14- 4 Testing flags to get conditions of jumps

CDT	Description / specification	Flag tested
-	Unconditional jump	none
EQ	ALU equal/zero	AZ=1
NE	ALU not equal/not zero	AZ=0
UGT	ALU unsigned greater than	AC=0 and AZ=0
UGE/CC	ALU unsigned greater than or equal	AC=0
ULE	ALU unsigned less than or equal	AC=1 or AZ=1
ULT/CS	ALU unsigned less than	AC=1
SGT	ALU signed greater than	AN=AV and AZ=0
SGE	ALU signed greater than or equal	AN=AV
SLE	ALU signed less than or equal	AZ=1 or AN<>AV
SLT	ALU signed less than	AN<>AV
MI	ALU minus/less than	AN=1
PL	ALU positive/greater or equal	AN=0
VS	ALU has overflowed	AV=1
VC	ALU has not overflowed	AV=0
MEQ	MAC or MUL Equal	MZ=1
MNE	MAC or MUL Not equal	MZ=0
MGT	MAC or MUL greater than	MN=0 and MZ=0
MGE/MPL	MAC or MUL positive or zero	MN=0
MLE	MAC or MUL less than or equal	MN=1 or MZ=0

MLT/MMI	MAC or MUL negative or less than	MN=1
MVS	MAC was saturated	MS=1
MVC	MAC was not saturated	MS=0

Flags used in Table 14- 4 are:

Table 14- 5 Flag specifications

Flag	For signed computing, flag is 1 when	For unsigned computing, flag is 1 when
AZ	ALU result is zero	ALU result is zero
AN	ALU result is negative	(ALU result has MSB set)
AC	ALU saturated (or carry out)	ALU carry out/borrow
AV	ALU result overflowed	no meaning
MZ	MAC result is zero	No unsigned computing in MAC
MN	MAC result is negative	
MS	MAC result was saturated	
MV	MAC result overflow (>40 bits), sticky	

Jump taken is a STATE of the PC FSM. The conditions of "Jump taken" include: (1). "It is a jump instruction" and (2). "the computing result on the selected flags is true". Jump taken means to load the PC with the target address in the program memory when there is a jump and the jump condition is true.

14.2.3.1 Design for unconditional jump

As soon as an unconditional jump instruction is recognized during the instruction decoding, the fetch of the next instruction and the flush of the fetched instructions (after the branch) will be decided/executed. However, a normal pipeline processor does not know if the instruction is a branch right after fetching.

Example 14-7 Analysis on execution of a unconditional jump in a 5-pipeline processor

```
      XXX
A:    JUMP B
      YYY
      ......
B:    ADD R1 R2
```

The execution details of the unconditional jump in this example is exposed and analyzed in the following pipeline execution table.

Table 14- 6 Unconditional jump pipeline execution table

clock	Pipeline stage 1	Pipeline stage 2	Pipeline stage3	Pipeline stage 4	Pipeline stage 5
	Instruction pointed	Instruction fetched	Instruction decoded	Operand fetched	Instruction executed
1	XXX				
2	JUMP to B	XXX			
3	YYY	**JUMP to B**	XXX		
4	B: ADD R1 R2	Flush YYY	JUMP to B	XXX	

5		ADD R1 R2	NOP(YYY)	JUMP to B	XXX
6			ADD R1 R2	NOP(YYY)	JUMP to B
7				ADD R1 R2	NOP(YYY)
8					ADD R1 R2

Planning for *pipeline execution table* is the basic method and tool for control path design. It exposes processor execution details in each pipeline stage, relations between pipeline stages, pipeline stalls, and hazards. Pipeline stages listed in the table is specified and illustrated in following Figure 14-13. By using the table, a designer can get execution details and the right time to apply controls. Thus, what happens right after the instruction fetch is that, *for example in clock 3, unconditional jump is decoded in pipeline stage 2*. The decoding (combinational logic output) can be used to control functions in the current clock cycle 3 (*for example: send jump target address to the Next-PC input using not latched control signals*) and to control functions the next clock cycle 4 (*for example: fetch instruction pointed by jump target B, flush instruction YYY, using latched control signals*). A 5-pipeline stage processor hardware associated with the table 14-6 is illustrated in the following Figure 14-13.

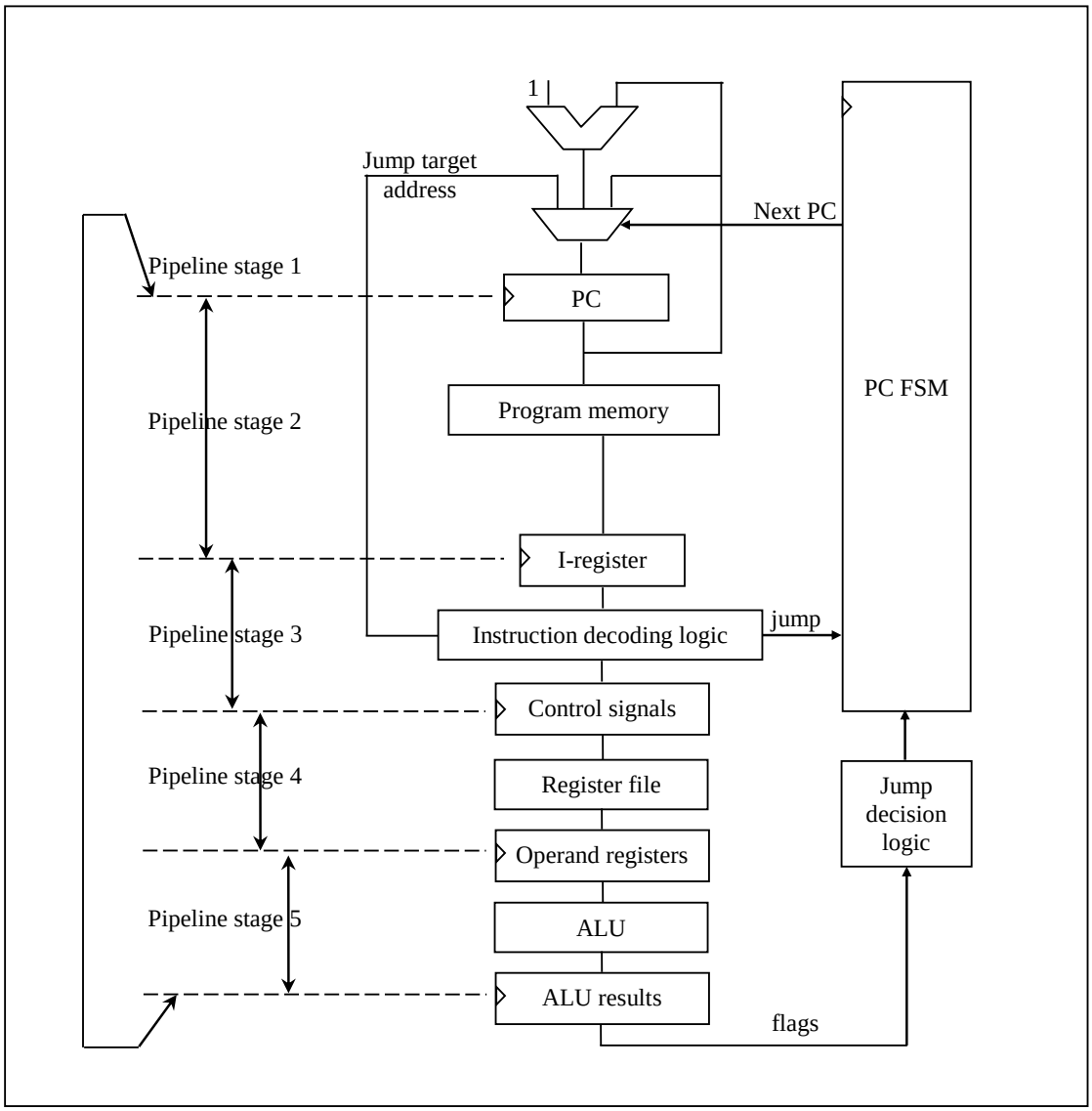


Figure 14-13 Simplified PC FSM in a 5-stage pipeline processor

14.2.3.2 Design for conditional branch

In a 5-stage pipeline processor, the jump taken (not taken) decision for a conditional branch (jump) will not be made until the previous instruction (before the branch) is executed in the last pipeline stage (pipeline stage 5 in our example). The flags for branch decision are available at the output of the result register. The processor has to keep loading of instructions after fetching the conditional branch and has to execute each pipeline functions of these fetched instructions until the branch taken/not-taken is known.

As soon as the conditional jump is taken, all fetched instructions after the conditional branch shall be flushed. The instruction pointed by the branch target address shall be fetched and be executed. In this case, 3 clock cycles are lost for each conditional jump taken; the processor performance is significantly reduced when many conditional branches are taken.

Example 14-8 Analysis on execution of a conditional jump in a 5-pipeline processor

```
     Move constant=1 to R1
     Move constant=1 to R2
     NOP //in case there is no register forwarding technique
     Compare R1 R2
A:   CJMP B GE // jump to B when great or equal
     INC R1
     INC R1
     INC R1
     UUU
     VVV
B:   ADD R1 R2
     ZZZ
```

Obviously, the conditional jump will be taken and the correct result should be R1= R1+R2=2. However, there are three instructions fetched and to be executed in the pipeline processor. We must flush these instructions so that we can keep the result being correct.

Table 14- 7 Conditional jump execution table when jump is taken

clock	Pipeline stage 1	Pipeline stage 2	Pipeline stage3	Pipeline stage 4	Pipeline stage 5
	Instruction pointed	Instruction fetched	Instruction decoded	Operand fetched	Instruction executed
1	Compare R1 R2	---	---	---	---
2	CJMP B GE	Compare R1 R2	---	---	---
3	X: INC R1	CJMP B GE	Compare R1 R2	---	---

4	Y: INC R1	INC R1	CJMP B GE	Compare R1 R2	---
5	Z: INC R1	INC R1	INC R1	CJMP B GE	**Compare R1 R2**
6	B: ADD R1 R2	Flush Z INC R1	Flush Y INC R1	Flush X INC R1	CJMP B GE
7		ADD R1 R2	Z=NOP	Y=NOP	X=NOP
8			ADD R1 R2	Z=NOP	Y=NOP
9				ADD R1 R2	Z=NOP
10					ADD R1 R2

And the hardware for this example is illustrated in the following Figure 14-14. When decoding the conditional jump instruction, the destination address is decoded and stored in control register CR1. It passed to control signal register CR2 in the next clock cycle and passed to CR3 after the next cycle.

When the result with flags are available from running the instruction Compare R1 R2, the jump decision logic and the PC FSM give the "jump taken" control signal, and performs following actions:

1. At the same clock cycle (cycle 5 in table 14-7), Not latched controls include selecting the next PC <= Jump target address, send Flush X, Flush Y, and Flush Z
2. At the next clock cycle (cycle 6 in table 14-7), The first instruction at the parget address is pointed. Instruction Y and Z are flushed and NOP are loaded to the instruction decoder instead of Y:INC R1 and Z:INC R1. The decoded outputs are replaced by NOP instead of Y:INC R1 and Z:INC R1, the operand output on pipeline stage 4 is NOP instead of operands for X:INC R1.
3. After the next clock cycle (cycle 7 in table 14-7), the instruction at the target address is fetched, and this instruction will be executed after three wasted clock cycles.

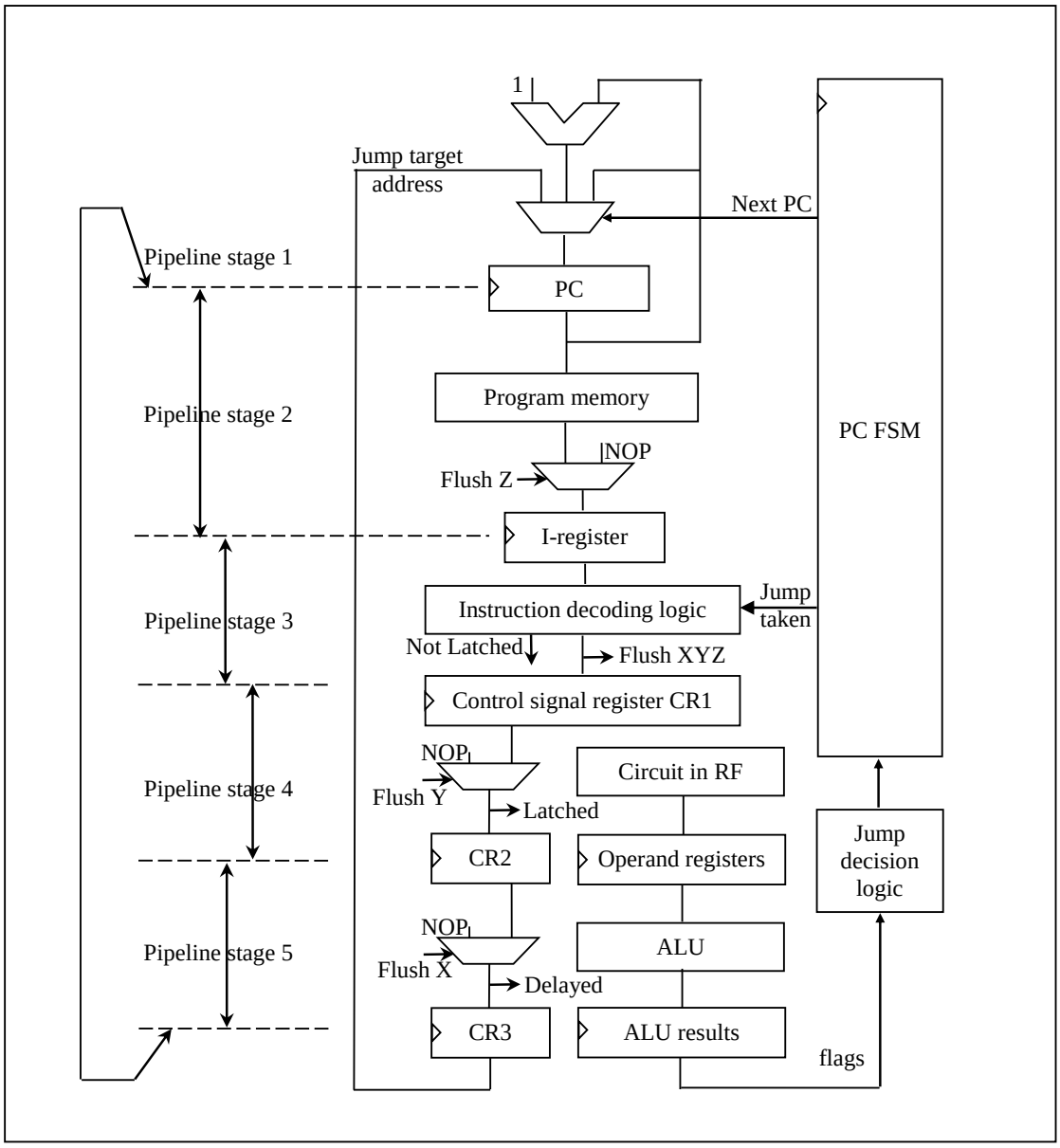


Figure 14-14 Simplified PC FSM in a 5-stage pipeline processor

14.2.3.3 Do not waste clock cycles while jumping

Classically, to easy programming, discussed control hazard handlings are designed in processor hardware, flush control is generated while a jump is taken. One/two clock cycles are wasted while running an unconditional jump. Three or more clock cycles are wasted when a conditional jump is taken. Jump instructions are frequently used in most programs. The wasted cycles are significant and the performance is thus decreased.

To keep performance, there are ways to minimize the cycle loss of jumps. An efficient way is not design hardware flush and handle branch (control) hazard in software.

The principle is:

1. There is no flush hardware, all flush and dependent control are in software (or in compiler).
2. If there are independent instructions to the branch instruction, re-schedule and run independent instructions before knowing the jump decision. It means, in the example 14-8, three independent instructions can be scheduled after a conditional jump.
3. If there is no (or not enough) independent instruction around the jump, inserted NOP should be followed to the jump instruction.

Senior processor, the example processor through the book, offered the flexibility to programmers to insert independent instructions or insert hardware NOP. A conditional jump instruction of Senior is thus coded as:

Jump [.*cdt*] [ds*x*] *target*

The operations are

If the condition .*cdt* is true, after *x* cycles: PC <- *target*

Example 14-9 Senior conditional jump

```
     jump.ne ds2 label4 // jump when not equal
     move r1, sr3 // ds2 means two instructions following the
     set  r2, 7   // jump will be executed anyway
     move r12,r3 // it will not be executed if jump is taken
label4
     set  r7, 3
```

The pipeline execution table of the example is:

Table 14- 8 Conditional jump running in Senior

clock	Pipeline stage 1	Pipeline stage 2	Pipeline stage3	Pipeline stage 4	Pipeline stage 5
	Instruction pointed	Instruction fetched	Instruction decoded	Operand fetched	Instruction executed
1	Jump.ne ds2 L4	---	---	---	---
2	Move r1, sr3	Jump.ne ds2 L4	---	---	---
3	Set r2, 7	Move r1, sr3	Jump.ne ds2 L4	---	---
4	Move r12, r3	Set r2, 7	Move r1, sr3	Jump is taken	Condition ready
5	L4: set r7, 3	Move r12, r3	Set r2, 7	Move r1, sr3	Jump.ne ds2 L4
6		L4: set r7, 3	Move r12, r3	Set r2, 7	Move r1, sr3
7			L4: set r7, 3	Flush if jump, otherwise move	Set r2, 7
8				L4: set r7, 3	NOP if jump, otherwise move
9					L4: set r7, 3

To conclude:

If ds*0*, all 3 following instruction will not be executed if the jump is taken (flush all).

If ds*1*, the first instruction following the jump will be executed anyway; and the second and the third following instruction will be flushed if the jump is taken.

If ds2, the first two instructions following the jump will be executed anyway; and the third following instruction will be flushed if the jump is taken.

If ds3, all three instructions following the jump will be executed anyway.

If the jump is not taken, there will be no flush.

14.2.4 Loop Controller

14.2.4.1 CONV loop controller

We discussed unconditional and conditional jumps. In a DSP processor, iterative computing are the dominant algorithms to be executed. In most cases, we execute FOR i=o to N-1 for iterative computing. In chapter 5, junior, we experienced the extra cycle cost while running a loop without repeat (CONV) instruction. Table 5-17 expose the difference of the cycle cost running a 40-tap FIR with / without hardware loop. It costs 7492/893 = 8.4 times more clock cycles using conditional jump while executing the algorithm. Therefore, we need hardware loop to speed up DSP.

A loop controller is a hardware module in control path of an ASIP/DSP processor supporting iterative computing to minimize cycle cost overheads. It (1.) counts the number of iterations, (2.) checks the finish of a loop and (3.) to terminate a loop. It runs a loop without extra cycle cost.

When running an iteration loop, the loop controller will decide the time to fetch the next instruction after the loop instruction, instead of running jump instruction multi times. Extra cycle cost induced by pipeline stall can therefore be avoided. While running the last step of the iteration loop, the loop controller decides to leave the loop and to fetch the next instruction after the loop instruction.

The loop controller is a sub-module inside the program flow controller. There are two kinds of hardware loops. The simple hardware loop supports loops consisting only a single-instruction. A simple loop controller is depicted in Figure 14-15. A simple loop controller consists of a loop counter and loop flag circuit. The simple loop controller counts the loop execution and sends flag when a loop is finished.

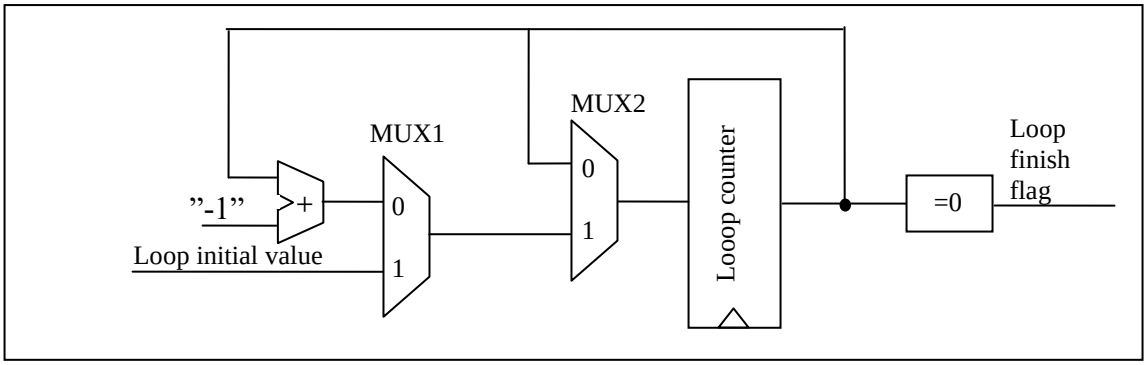


Figure 14-15 Single instruction loop controller

In Figure 14-15, the simple loop controller supports single instruction loop such as convolution instruction. This loop controller can be configured by writing the length of a loop to the loop counter. A loop controller keeps down-counting while running the loop instruction. When reaching zero, a loop finish flag will be set.

In Figure 14-15, the initial value to the loop counter is loaded by setting "MUX1=1" and "MUX2=1". The loop counter keeps its value before starting the loop by having "MUX2=0". While running the loop instruction, the loop counter keeps down counting until it reaches zero (or reaches 1 to match control pipeline).

14.2.4.2 REPEAT loop controller

Another loop controller, REPEAT loop controller, supports hardware loop for multi instructions (including single instruction loop). The REPEAT instruction could be, for example, **REPEAT M N**. It carries M instructions in a loop and run the loop N times. Two hardware counters are required, one is MC (the inner loop program counter, with initial value M from the REPEAT instruction) which counts the steps of inner loop, and the other one is LC (the loop counter, the number of iteration of the loop, with initial value N from the REPEAT instruction). The behavior of the REPEAT loop controller is described by the following pseudo code:

```
  /* In Hardware */
RL=M;
LC=N;
  /* M is the loop-code-length register, N is the loop-length */
  /* RL points the remaining lines to be executed in the loop */
  /* LC is the loop counter */
  LoopStart = PC; /* Keep the start point of the REPEAT */
IL:
  if(RL !=0)
  {
    RL = RL - 1;
    PC = PC + 1;
    goto IL;
  }
  else if(LC!=0)
  {
    LC = LC - 1;
    RL = M;
    PC = LoopStart;
    goto IL;
  }
  else LoopFlag = 1;
```

Following the behavior description, the PC FSM should be modified and redraw in Figure 14-16.

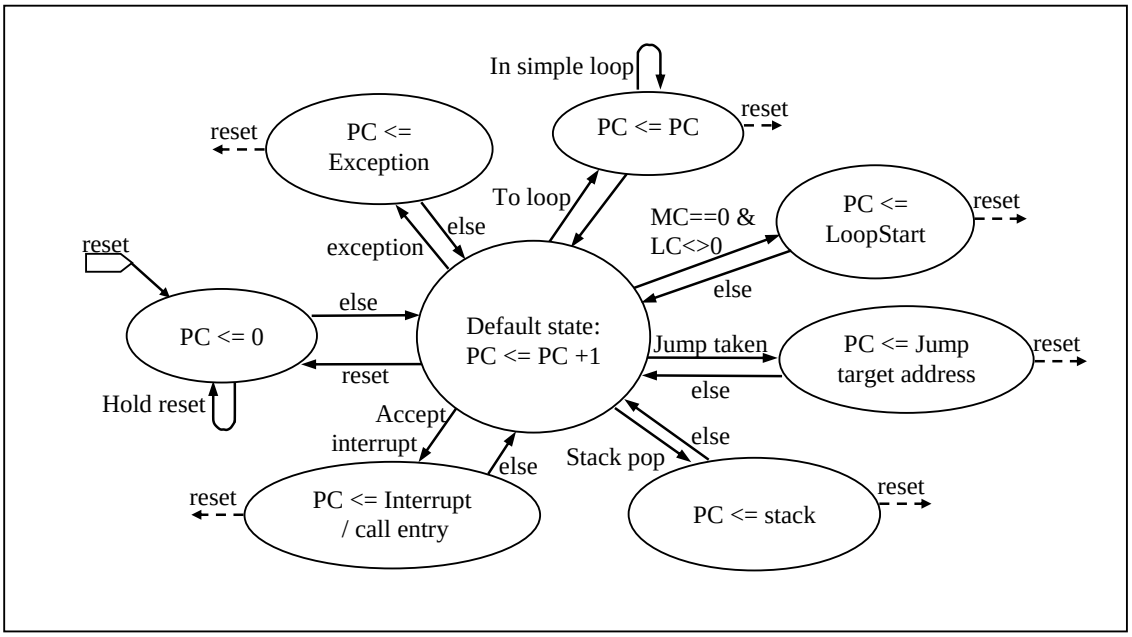


Figure 14-16 PC FSM supports REPEAT

The circuit schematic of a REPEAT loop controller is:

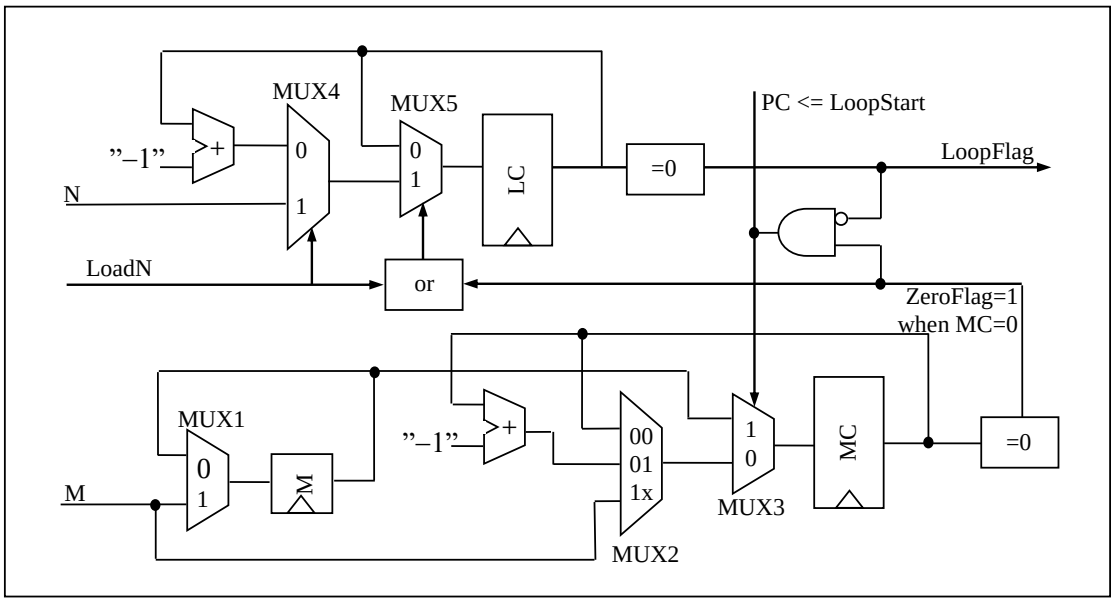


Figure 14-17 Loop controller for advanced hardware loop

M is loaded when MUX1=1. MC down counts while running REPEAT. When MC=0 and LC<>0, the M will be loaded to MC again, PC<=loopStart, and MC starts down counting again. N is loaded when LoadN=1. NC down counts when ZeroFlag is "1". When LC is "0", the LoopFlag is set.

14.2.5 Priority in PC FSM

For some specific ASIP, the PC FSM must be implemented following predefined

decision priority. High prioritized conditions shall mask low prioritized conditions. Carefully planning the prioritized decision order can avoid design bugs in the PC FSM. The decision order should be specified before the implementation of a PC FSM. The RTL coding must therefore be in a right way. For example, if the decision order is specified, the IF-THEN-ELSE coding style should be used and using of CASE should be avoided.

```
if(reset) begin
  // React on reset
end else if(debugging_mode) begin
  // Decoding and supporting the debugging mode
end else if(exception) begin
  // Decoding for exception handling
end else if(interrupt) begin
  // Decoding for interrupt
end else begin
  // Decoding the fetched instruction
end
```

14.2.6 Exception, Interrupt, jump, Conditional Execution

When an exception happens, instructions in all hardware pipeline stages shall be flushed. It is not necessary to keep running the application codes after an exception because the execution of the current task is already wrong.

When an interrupt is taken, the PC and the current flags must be stacked. At the same time, the interrupt service entry is loaded to the PC. The interrupt in PC FSM shall be designed as a jump plus pushing PC to stack. Before serving an interrupt, both PC and current flag values shall be pushed in stack.

There could be differences between a return from a function CALL and a return from an interrupt. When executing a return from interrupt, EOI (End Of Interrupt) signal shall be generated indicating the differences. When running a return from a function call, the stacked PC should be popped from the stack. However, when running a return of interrupt, both PC and flags must be returned from the stack. Flags of the CALL function result might be pushed/popped by programmer.

If the condition is true while running a conditional execution, the result should be written back, otherwise, the execution will be ignored. When running a conditional execution instruction, the signal "condition is true" should be used as the enable signal to all modules which can accept the result of conditional execution. Typical modules are general register file, data memories, and PC FSM.

14.3 Instruction Decoder

14.3.1 General

An instruction decoder decodes the fetched instruction and generates control signals. The control signals are divided into control signals towards the datapath (controlling

ALU, MAC, register file, and special registers), the control signal towards data accesses (for memory control and address generation), and control signals inside the control path (for program flow control).

14.3.2 Control Signal Decoding

Figure 14-18 illustrates an instruction decoder based on decoding logic circuit. It decodes machine codes directly into control signals. The decoding logic is usually based on AND-OR logic. Part of AND signals could be inverted. Each AND-OR circuit generates one control signal. A control signal can be decoded from several instructions using OR function. A sub coding field from an instruction (including inverted and not inverted signals) comes to one AND gates.

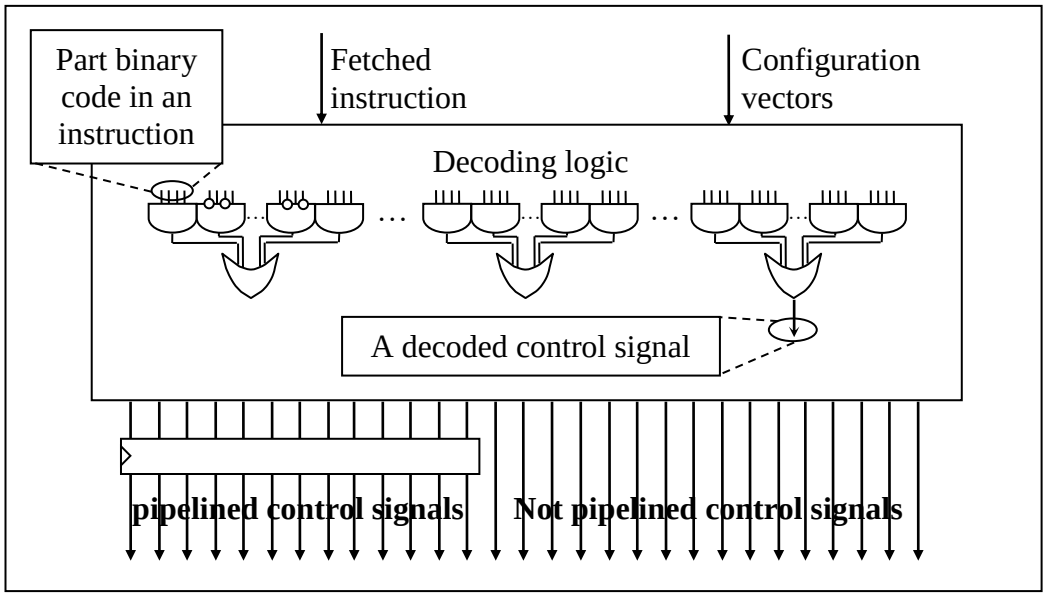


Figure 14-18 Instruction decoder using custom logic

According to the pipeline design, control signals for each pipeline shall be available at the specific pipeline stage. Control signals, to be used as soon as possible during instruction decoding, will be connected and used right after decoding and not latched, see "Not pipelined control signals" in Figure 14-14. Control signals, to be used after the decoding stage, will be connected and used after the pipeline register, see "Pipelined control signals" in Figure 14-14. Control signals, to be used even later, will pass more pipeline registers, see signals "Delayed" in Figure 14-14.

A control signal might be driven by many instructions and one instruction will drive many control signals. A control signal towards a bus will drive every bit on the bus, which means that the fan-out of a control signal could be very large. One functional control signal may drive hundreds of input pins. In chapter 11, a typical fan-out of a control signal was analyzed by an example. 512 fan-outs were identified. To manage huge fan-out on physical level, image signals can be used to share the load. By using image signals, huge fan-out can be physically distributed to several physical drivers. For example, an output signal from one decoding logic block can be split and

assigned to multiple modules (if the synthesizer cannot do it):

```
control_signal_100 = decoding_logic (condition1 … condition n);
control_signal_100_ALU = control_signal_100; // split it to ALU
control_signal_100_MAC = control_signal_100; // split it to MAC
control_signal_100_MEM = control_signal_100; // to memories
```

14.3.3 Instruction decoding for Register forwarding

Control for register forwarding is to avoid the RAW data dependency. In this case, the following instruction uses the result from previous instruction and the result is not yet written back. The implementation of register forwarding shall be for both the hardware and the instruction set simulator. In control path, each arithmetic and logic instruction shall have an execution cycle cost indicator. When the execution cycle cost is one cycle, only the following instruction will be associatively decoded. When the execution cycle is N, N following instructions will be associatively decoded. The following table exposes the decoding association for register forwarding:

Table 14- 9 Register forwarding control association

clock	The execution cycle cost by the previous instruction		
	1	2	3
1	Start execute previous instruction in datapath		
2	The following instruction OPx forward control =1 when OPx should be forwarded	when OPx should be forwarded, hold the 1st following instruction, or run independent instruction	when OPx should be forwarded, hold the 1st following instruction, or run independent instruction
3		The 1st following instruction OPx forward control =1 when OPx should be forwarded	when OPx should be forwarded, hold the 2nd following instruction, or run independent instruction
4			The following instruction OPx forward control =1 when OPx should be forwarded

If the programmer realizes the holding cycle cost induced by the multi-cycle RAW dependency, the RAW shall be avoided. Programmers shall insert instructions that their operands are independent to previous instructions (with multi-execution-cycle).

14.3.4 Issues of Multi-cycle Execution

There are instructions consuming multiple execution cycles. For example a MAC instruction takes two execution cycles; one for multiplication and one for accumulation. Special decoding might be needed for example for register forwarding. The decoding principle for the MAC execution is, for example:

- During the first cycle, the multiplier executes the multiplication,
- The accumulation should be hold without doing anything if the previous instruction is a single cycle instruction,
- If the previous instruction is another MAC, the accumulator should execute the accumulation for the previous instruction.
- During the second execution cycle:
- The multiplier should be released and can be used by the next instruction,

- The accumulator works for the current MAC instruction: adding the result from the multiplier to the accumulator.

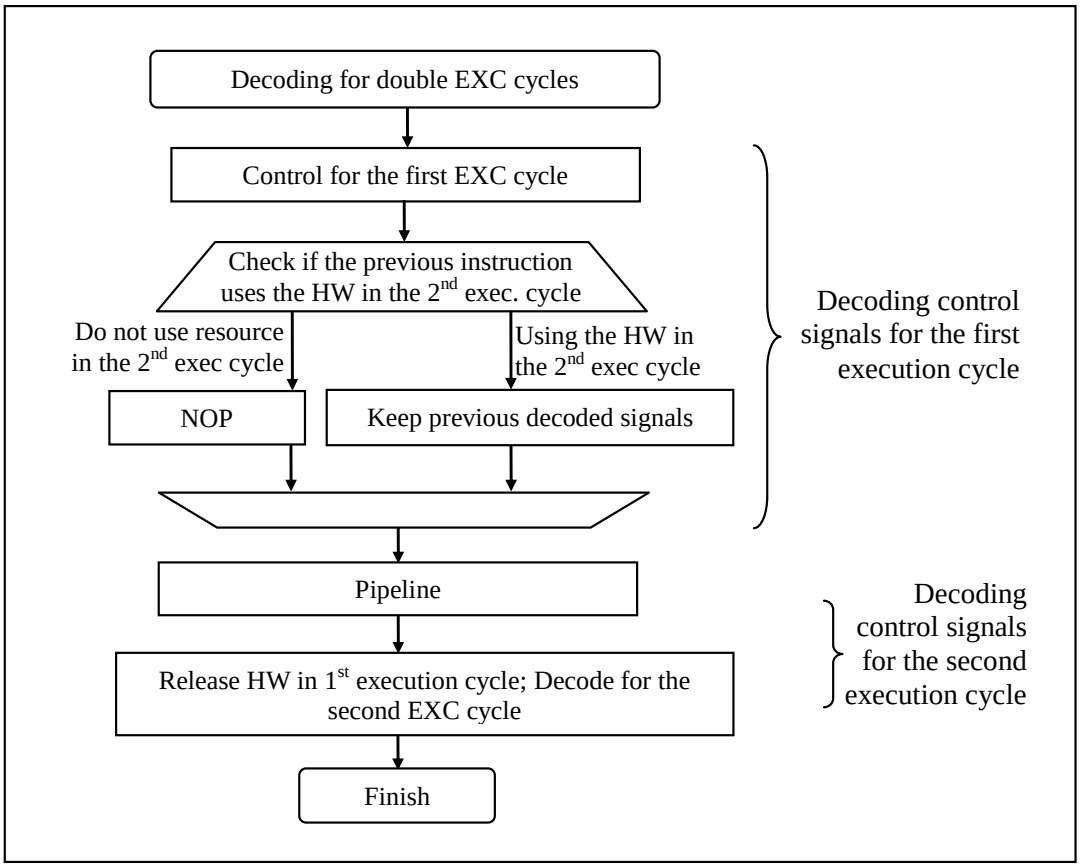


Figure 14-19 Decoding for double (execution) cycle instructions

If the MAC result is needed by the following instruction, the register forwarding method described by Figure 14-11 cannot be used and a data dependency hazard occurs. There are different ways to handle this hazard. One way is to design a hazard dependent checker in the instruction set simulator (check the lab work), if there is data dependent in firmware, warning or error will be reported and the firmware designer takes care of the hazard by inserting independent instructions or NOP. It may not be sufficient because the data dependence may not happen during simulation time. Hardware hazard checker might be useful. The hardware hazard checker will insert a NOP by the way illustrated in Figure 14-14 when this kind of data hazard happens.

Example 14-10 Design pipeline for the instruction decoder of the "Senior" processor core.

According to the pipeline design shown in Figure 14-10, the pipeline design of the instruction decoder is given in Figure 14-20.

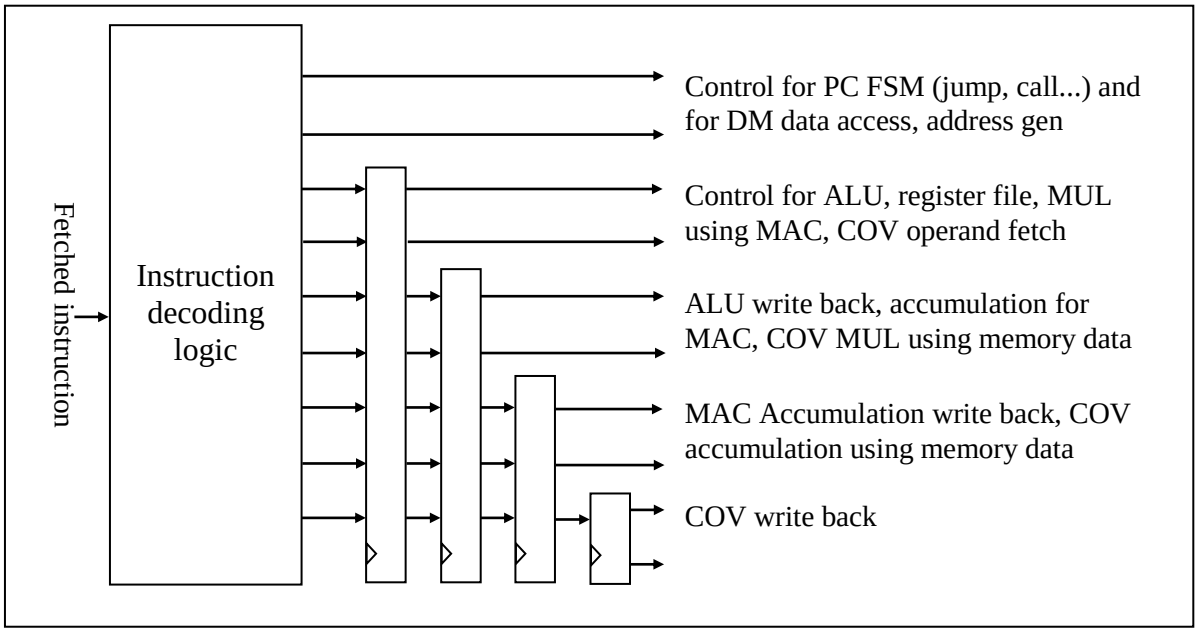


Figure 14-20 Pipeline design for Senior instruction decoder

14.3.5 VLIW machine decoding

A VLIW processor consists of multi datapath modules. Each datapath module is controlled by a specific instruction, a part of a long instruction word. While fetching a long instruction, the long instruction is split into several short instructions. Each short instruction is assigned to a specific datapath module. A datapath module can be an ALU, a MAC, a register file, a data memory, or a branch unit.

Write hazard is a structural hazard writing multiple data into one register at the same time. It is usually handled by compiler. It can also be handled during the hardware design by splitting a long register file into several cluster register files and assigning each register file to a datapath cluster. Data in all clusters is globally available all the time as operands. Each cluster generates only one result at a time and a result of a cluster can only be written to its own local register file.

Read dependency is to read a value which is not yet valid (as a result in order). It was handled during the code compiling time or code simulation time. In this case, the programmer or compiler must identify the read dependency and avoid accessing and using the dependent data by re-scheduling execution. If the code is written by hand, the data dependencies should be identified by the instruction set simulator.

14.3.6 Decoding for Superscalar

To enhance performance, a superscalar handles data dependencies dynamically by special hardware during run time. Multiple instructions are fetched for executions. Independent instructions can be executed OOO (Out Of Order) to minimize

hardware waiting time. Results are released to register file following execution order through reorder buffer. Therefore, data hazard can be dynamically handled and a fetched instruction can be executed as soon as operands are available (as a results of another instruction).

The write dependency is handled in reorder buffer, which is the hardware unit releasing results in programed order to register file. The read dependency protection must be handled in *both* instruction dispatch unit *and* the reservation station. All dependencies are handled by checking the instruction fetching order and managed via instruction renaming. The instruction renaming is to change an instruction fetching name (PC value) to a short name indicating fetching order. The fetching order will be used to set up dependency relations. Dependencies can be handled, for example, by the following the rules:

- When multiple writes occur to the same register, the result of the latest fetched instruction has the right to write.
- When an operand is the result of an early fetched yet not executed instruction, the operand fetching cannot be executed until the instruction is executed.

This kind of decoding is called interlocked-decoder, a decoder with dependency tolerant hardware, in superscalar. It decodes an instruction according to fetching order and execution status of other instructions. If there are dependencies, the decoded instruction will not be executed until the interlock is released. The principle of superscalar was discussed in chapter 3. The instruction decoding and dependency management hardware in a superscalar can be found in Figure 14-21.

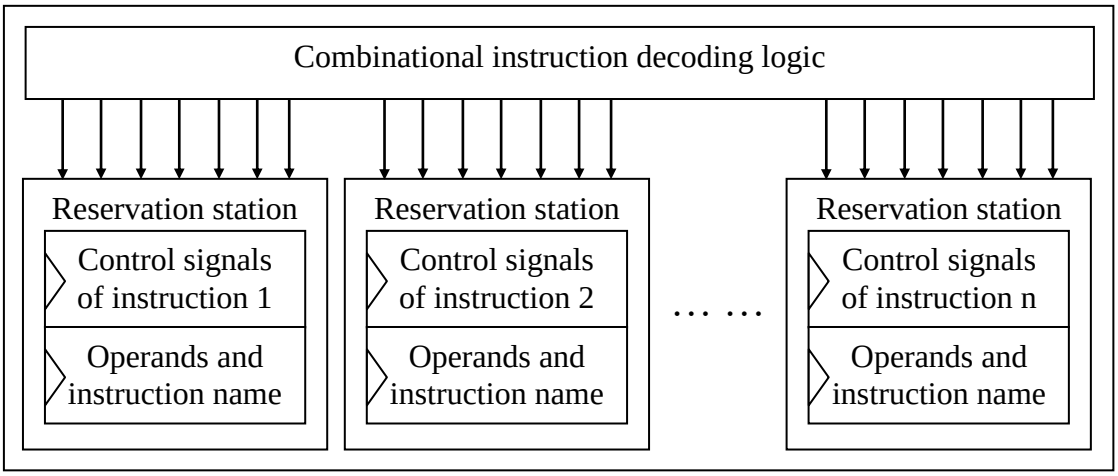


Figure 14-21 Instruction decoding for HW with dynamic pipeline

By using reservation stations in a superscalar, decoded control signals, operands, and operand addresses are reserved in one reservation station temporally named by the name of an instruction. Operand dependencies are checked by hardware. When there are no data dependencies, operands can be fetched and the instruction can be issued to an execution unit together with the decoded control signals.

14.4 Control Path Hardware Integration

14.4.1 Top-level Structure

A simplified control path block diagram in a DSP core is illustrated in Figure 14-22.

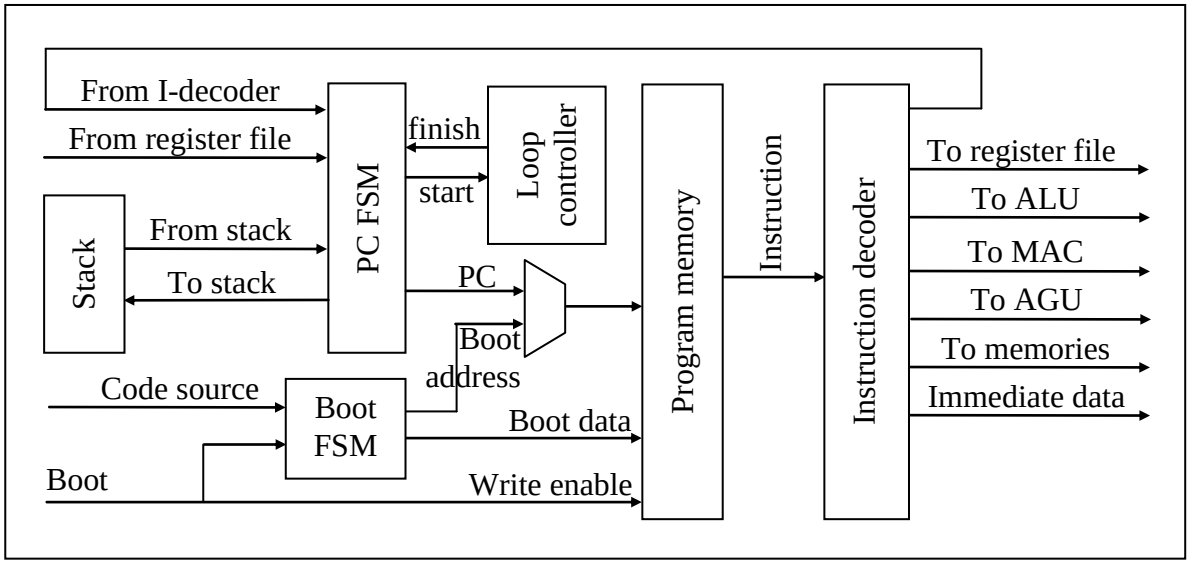


Figure 14-22 Simplified control path diagram

The program memory stores binary program codes to be executed. Code are loaded either by a code-loader or by master MCU via DMA during the system initialization phase. The program memory could be booted either during the reset (power-on) process or during the initialization of a new application. The booting FSM must be independent to the instruction flow controller because the instruction flow controller does not work until the code is loaded.

Processor configuration vectors are stored in control and status registers. The running program can configure the DSP processor to get certain processor features for specific applications. For example, a processor can be configured into the unsigned mode, signed integer mode, saturation can be enabled after certain instructions, certain peripheral ports can be enabled, etc. Configurations are performed by assigning configuration vectors to configuration registers.

A hardware stack buffer can be designed in the control path in some ASIP. It stores the PC value and flags as the return information of a fast interrupt. When executing CALL-RETURN, only PC instead of flags will be stored. The stack in the control path is only for the program flow control.

14.4.2 Program Memory Design

In most cases, the program memory is SRAM which stores binary codes being booted from main memory (or ROM) outside the DSP chip during the power-on procedure or during an initial phase for a new application. In special cases, a program memory

could also be a hard coded ROM and fixed during backend design.

The PC width, the number of bits in a PC, defines the maximum program memory size. For example, if the program memory size is 256k words, the length of PC should be 18 bits (2^{18}=256k). It is good if the PC width of an ASIP is the same or less than the native data width. For example, the native data width is 16b and the width of PC is also 16b. If the PC width is larger than the native data width, a register value in the general register file will not cover the code address space. A general register will not be directly used as a target address of a global jump. The firmware design will be complicated.

14.4.3 Loading code

A booting-FSM Loads codes (program in binary format) to PM in the DSP. The booting FSM could be a peripheral device (independent to the ASIP/DSP core) or a module in control path besides the PC FSM. Since the boot FSM could be tightly connected to PM, it will be briefly discussed here in this chapter.

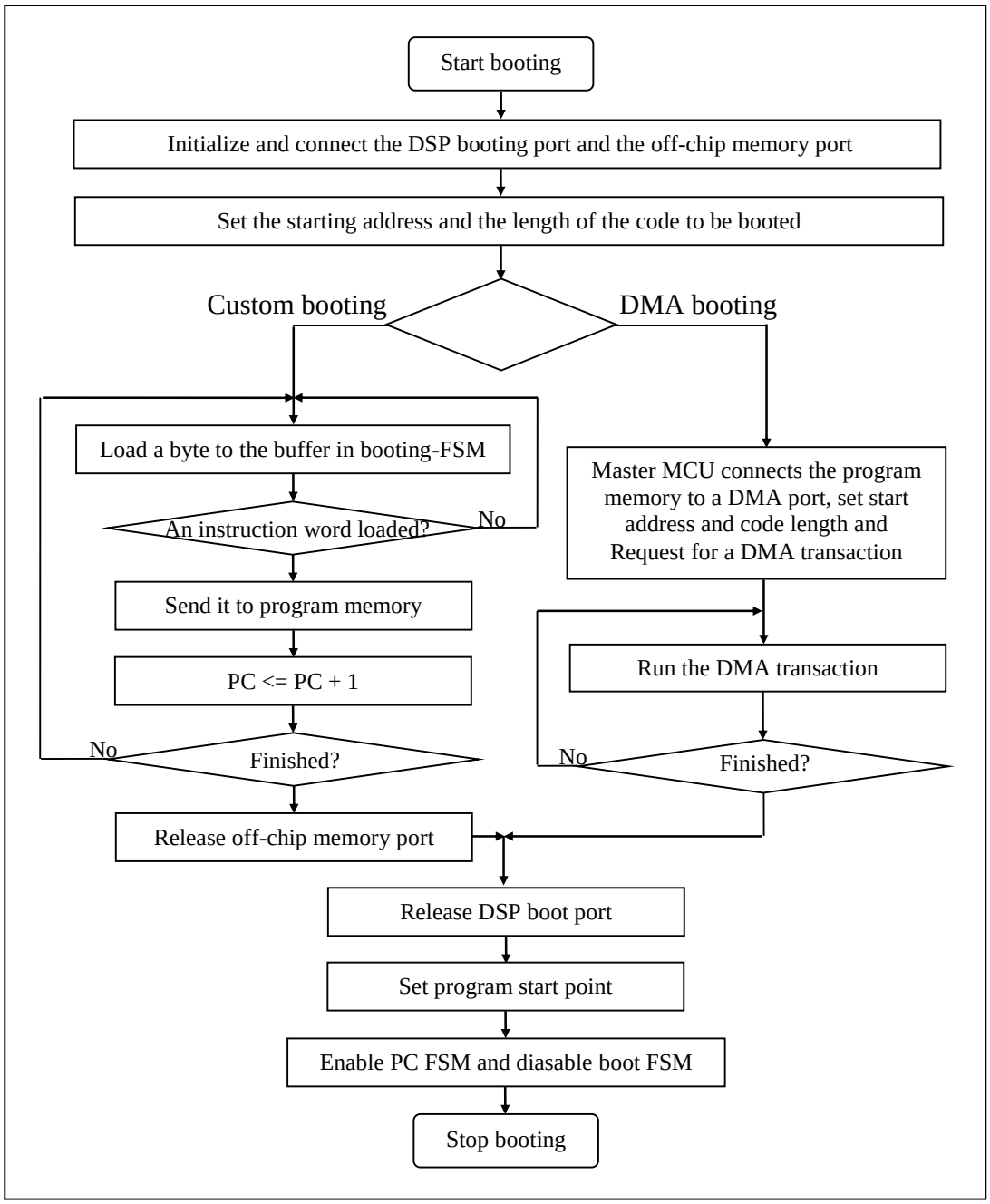


Figure 14-23 Example of program load at boot time

Normally, the application program is stored in off-chip memory, a ROM (Read Only Memory) or a DRAM. Three phases of booting should be managed by the booting FSM:

1. Initialize ports and devices involved in booting
2. Load initial program into program memory
3. Finish a booting by releasing booting resources and start the PC FSM.

And the DSP (ASIP) processor may boot rest of the (other) code using booted codes.

During the initialization phase, the booting FSM first initializes a special program memory write port (data and address) in the DSP (ASIP) and connect the port to the

off-chip memory. The booting FSM also initializes the off-chip memory, its port and configures its access mode. The program memory port and the off chip memory ports are thus connected (with or without a FIFO buffer). After assigning the initial address to the PC, the booting procedure starts.

Most ROMs supply data in byte format and the length of an instruction word could be much longer, 16-bit, 24-bit, 32-bit, or longer. During the program booting phase, the data type should be re-formatted to adapt to the instruction word width.

When the booting is finished, the ROM and the port for booting should first be disconnected. Before shutting down, the booting-FSM initializes the PC-FSM by assigning the PC starting address. The PC starting address points and loads the first line of code. The default PC starting address can be "zero".

DMA will be discussed in Chapter 16. If there is no booting FSM in a DSP processor, the micro controller (MCU) can boot programs for it. The program can also be booted by using JTAG machine (see more in Chapter 19). The booting procedure is roughly the same as the flow in Figure 14-23 except that the master is MCU instead of the Booting FSM. While booting, the MCU holds the DSP processor by locking or disconnecting the PC. After the booting, the MCU will write the initial address to the PC of the ASIP/DSP processor and release the PC or connect the PC to the ASIP/DSP processor, see the PM addressing port in Figure 14-5.

14.5 PC Stack

In a simple ASIP, we may need a PC hardware stack to minimize the code cost by running push and pop implicitly while running interruptions, CALL and RETURN. It is only used to stack PC and flags when supporting interrupts and procedure calls.

The block diagram depicted in Figure 14-24 is an example of a simple stack with four stack registers. It performs push and pop by changing the stack pointer instead of pushing or popping through the register chain one by one. Therefore, the power consumption of the stack can be low. To simplify the circuit, a one-bit stack is presented in Figure 14-24. If PC and the flags must be saved in one clock cycle the actual stack register width should be the width of PC plus the width of the flag register. The stack size (the depth) is the number of stack registers in the stack.

The control signal inside the stack is generated from the finite state machine FSM-SPR (Stack Pointer Register). The stack pointer up counts while pushing and down counts while popping. In the specific circuit in Figure 14-24, SPR[1:0] is used as the visible stack pointer and SPR[2] indicates the stack overflow. The stack is empty when SPR[2:0] = 000, the stack is full when SPR[2:0] = 011.

The left part of Figure 14-24 is the stack controller, the FSM, and the right part is the stack registers S1R, S2R, S3R, and S4R. In this Stack Controller, decoding of the control signals is based on the input signals, push or pop, as well the old SPR value. Control signal M1 to M4 are push control signals of S1R to S4R. Only one of them can be "1" at a time. The data will be pushed to S1R if M1 is "1". When M1 is "1", M2, M3, and M4 must be "0" to keep the values in S2R, S3R, and S4R. The pop control

signal is SPR[1:0] selecting one of the SxR to the pop out port.

When the stack is full and the stack receives a “push” request, an overflow will be reported. When the stack is empty and the stack receives a “pop” request, an underflow will be reported. An operation error will be detected when both "push" and "pop" are requested the same time. Both overflow, underflow, and OpError should generate an error/exception message.

After reset, the SPR (Stack Pointer Register) in the Stack Controller will be “000”, which points to the top of the stack waiting for data to be pushed in (nothing to pop out). When a “push” comes, M1 <= Push & (SPR = 000), S1R is selected to accept the data from the “push-in data” port. After this clock cycle, SPR =SPR +1=01. S1R is selected to the pop port after pushing data to S1R. When there is no push and no pop, SPR keeps its value. When a “pop” control comes, the S1R is connected to the “pop-data” by the control signal SPR[1:0]. After a “pop” operation, SPR<=SPR-1 and the stack will be empty again.

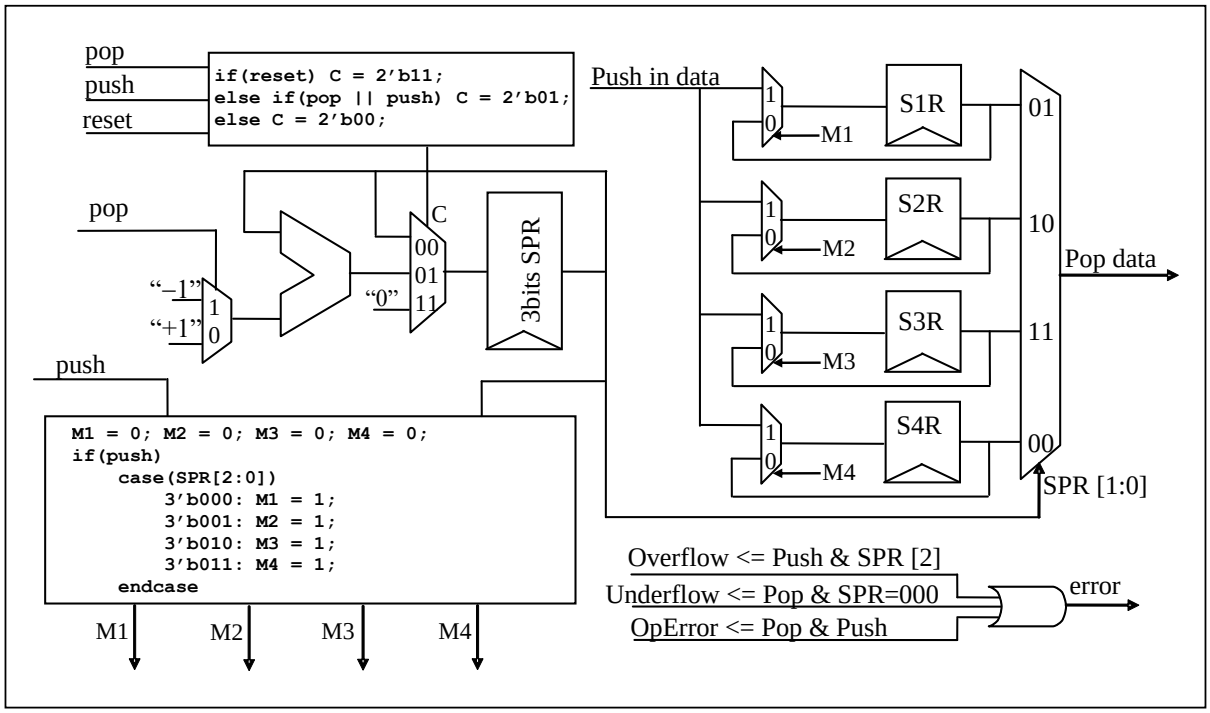


Figure 14-24 PC stack for fast interrupts

When another push comes after pushing data to S1R, S2R is selected to load because M2 <= Push & (SCR = 001). After pushing the value in S2R, the SCR become 10 and will point to S2R as the pop register. The SCR is updated (plus one or minus one) after a push or pop command. A stack access error occurs in the following cases:

Table 144-10 Possible stack operation errors

Error Type	Error Conditions
Illegal operation	The push and pop control signals are true at the same time.
Underflow	A pop occurs while all SxRs are empty (SCR = 000).
Overflow	A push occurs while all SxRs are used (SCR = 011).

14.6 Conclusion

The organization of control path function and the implementation of control path micro architecture have been discussed based on a five-pipeline ASIP/DSP processor. Fundamental knowledge was introduced in the control path organization part. The micro architecture implementation of a control path includes the design of the PC FSM and the instruction decoder. Loop controller and stack machine were also discussed.

14.7 References

[1] www.wikipedia.org
[2] Viktor Öwall, Synthesis of Controllers from a Range of Controller Architectures, Ph.D. Thesis, Department of Applied Electronics, Lund University, Dec. 1994. CODEN: LUTEDX/(TETE-1010)/1-170(1994).
[3] Eric Tell, Mikael Olausson, and Dake Liu, A GENERAL DSP PROCESSOR AT THE COST OF 23K GATES AND 1/2 A MAN-YEAR DESIGN TIME, ICASSP 2003, Hong Kong May 2003.

14.8 Exercises

1. What are main functions in a control path of a simple DSP/ASIP?

2. Describe functional blocks in Figure 14-22, e.g. the PM, the PC FSM, the instruction decoder, the stack, and the loop control.

3. Implement the program booting procedure in Figure 14-23 using pseudo code.

4. Design and implement a PC FSM using HDL including the circuit schematics, the jump taken decision circuits based on the flags from the ALU including "sign", "zero" and "saturation/carry out". The following cases of Next PC should be included:

a) PC <= PC + 1;

b) PC <= PC;

c) PC <= 0;

d) PC <= Target address; //Jump taken

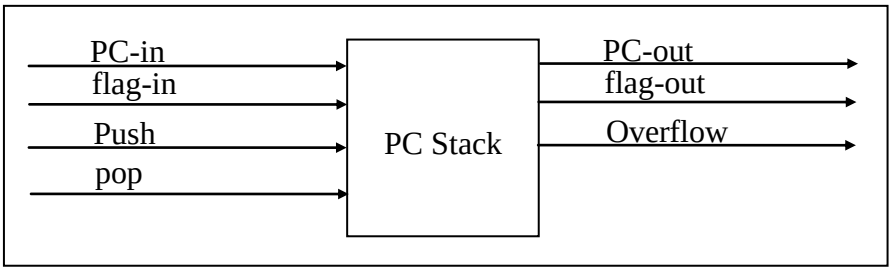
e) PC <= Stack pop;

f) PC <= Loop start address;

Jump taken happens when a conditional jump instruction is executed and the jump is taken.

5. Design a loop counter including initialization circuits. The output of the loop counter is the "loop finish flag"=1 when the loop counter counts down to "1" (next will be "0").

6. Design a hardware stack circuit with the depth of 8 storing PC and flags. PC_in[23:0] and flag_in[2:0].



7. How to decode control signals for multi-cycle instructions?