XM-23p – The XM-23 Pipeline Processor
Design Document

Larry Hughes, PhD
Electrical and Computer Engineering
Dalhousie University

19 May 2024

1 XM-23

The XM-23 processor is a "traditional" von Neumann architecture consisting of three distinct stages: fetch (the instruction is fetched from memory), decode (the instruction is separated into its opcode and operands), and execute (the CPU performs the instruction on the operands).

Most XM-23 instructions perform the three stages in four clock cycles (see Figure 1): two for the fetch accessing memory (steps F0 and F1); one to decode (step D0); and one to execute (step E0).

Clock	Instruction	Stages	Comments
N+0	`ADD R1,R2`	F0	Write PC to memory; Increment PC
N+1		F1	Read instruction from memory location
N+2		D0	Inst ← ADD; SRC ← R1; DST ← R2
N+3		E0	R2 ← R1 + R2; Update PSW

Figure 1: CPU-resident operations

However, some instructions require memory to be accessed to store or retrieve information. In these cases, the execution stage requires an additional clock cycle to access memory (i.e., read or write data), step E1 (see Figure 2).

Clock	Instruction	Stages	Comments
N+0	`ST R1,R2`	F0	Write PC to memory; Increment PC
N+1		F1	Read instruction from memory location
N+2		D0	
N+3		E0	MBR ← R1; MAR ← R2; CTRL ← WR
N+4		E1	Write MBR to MEM address in MAR

Figure 2: Memory-access operations

The XM-series of processors have been successful. Since the XM-series were intended for a RISC pipeline architecture, we are now developing a new version of the XM series, XM-23p, which uses the standard XM Instruction Set Architecture but built on a pipelined CPU.

This should increase instruction throughput, reducing the time to complete an instruction to two clock cycles (rather than four) and memory-accessing instructions to three clock cycles (rather than five).

The basics of the design are explained in this document.

2 Overlapping stages

A problem with the existing von Neumann architecture is that the stages are done sequentially. When the fetch has finished, the decode stage starts (fetch is idle), and when the decode has completed, execution begins but fetch and decode are idle.

An alternative is to perform the fetch, decode, and execute stages in parallel. The problem is, we cannot decode an instruction until is has been fetched, and we cannot execute an instruction until it has been decoded.

A solution is to decode and execute an instruction after it has been fetched, while at the same time, starting the fetch for the next instruction. In other words, the decode and execute of an instruction occurs at the same time as next instruction is being fetched.

Examples of memory-resident operations stages are shown in Figure 3 . (The `MOV` instruction follows the same sequence of steps, performing the decode at time N+4 and execute at time N+5.)

Clock	Instruction	Stages			Comments
		Fetch	Decode	Execute	
N+0	`ADD R1,R2`	F0	D0		
N+1		F1		E0	Previous instruction is completed.
N+2	`MOV R2,R0`	F0	D0		
N+3		F1		E0	R2 ← R1 + R2; Update PSW

Figure 3: Pipeline fetching, decoding, and executing an instruction

The stages are interpreted as follows (for details on steps F0 and F1, see section 3):

Step	Action
F0	• The address of the next instruction is supplied to instruction memory.
F1	• The instruction is returned from memory and placed in the Instruction Register.
D0	• Decode the instruction from the Instruction Register into instruction-specific fields. • The opcode for `ADD` is extracted and its operands (R1 and R2) are made available. • At time N+2, the F0 fetch stage of the `MOV` instruction begins.
E0	• Perform the required operation on the instruction-specific fields. The instruction is completed. • At `ADD` of R1 and R2 is performed; the PSW is updated; and the result is written to register 2. • At time N+3, the F1 fetch stage is completed and the `MOV` instruction is ready for decoding.

When a stage has finished, its results become the input for the next stage: fetch-to-decode and decode-to-execute.

It still takes four stages (two fetch, one decode, and one execute) to complete the instruction cycle; however, because of the overlapping stages, an instruction is executed every two clock cycles.

The XM-23p CPU can be represented as a state diagram, such as the one shown in Figure 4, where:

START: *START* is the initial state. The state transition occurs on a clock tick. The transition actions are to load the instruction register with a NOP and then start steps F0 and D0. The next state is *Wait tick 0*.

Wait tick 0: When the CPU is in state *Wait tick 0*, the next clock tick causes steps F1 and D0 to be performed. The next state is *Wait tick 1*.

Wait tick 1: At state Wait tick 1, the instruction has finished the Execute state (step E0). On the next clock tick, steps F0 and D0 begin. The next state is *Wait tick 0*.

The above steps repeat until an exception occurs (not shown).

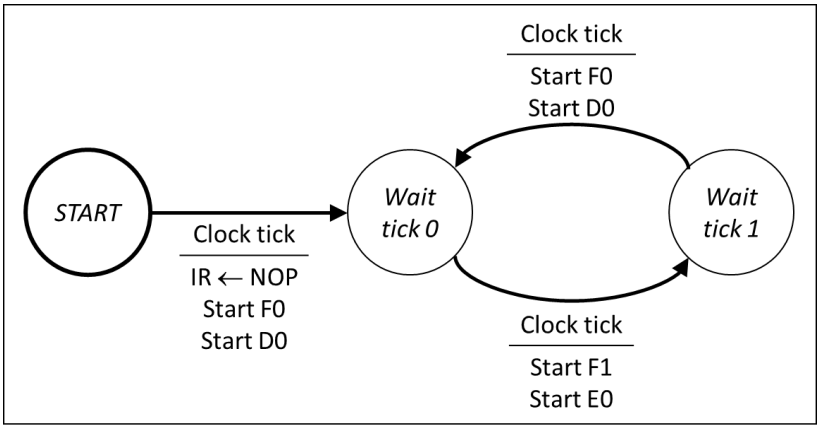

Figure 4: XM-23p CPU state diagram

The CPU state diagram is incomplete in that it does not support accessing data memory, when an extra execution stage, E1, is required (see section 5).

3 Instruction fetch

Instructions are fetched from the instruction memory, IMEM, during the Fetch stage's two steps, F0 and F1. IMEM is accessed using three registers (see Figure 5):

IMAR: Instruction Memory Address Register. Holds the address of the instruction to be fetched. Write only.

IMBR: Instruction Memory Buffer Register. Has the read from location specified in IMAR. Read only. (Alternatively, this can be referred to as the Instruction Memory Data Register or IMDR.) At the end of F1, the IMBR is copied into the Instruction Register (IR).

ICTRL: Instruction Memory Control Register. Indicates that an instruction is to be read from the address specified in the IMAR.

The Fetch steps use the registers as follows:

Steps	Action
F0	• The PC is written to the IMAR • READ is written to the ICTRL • The PC is incremented to the next address.
F1	• The instruction memory uses the address specified in IMAR and returns the contents of the location to Instruction Memory Buffer Register (IMDR), or Data Register (IMDR). • The ADD instruction fetch finishes at time N+1. • The fetched instruction is copied to the Instruction Register (IR) for decoding.

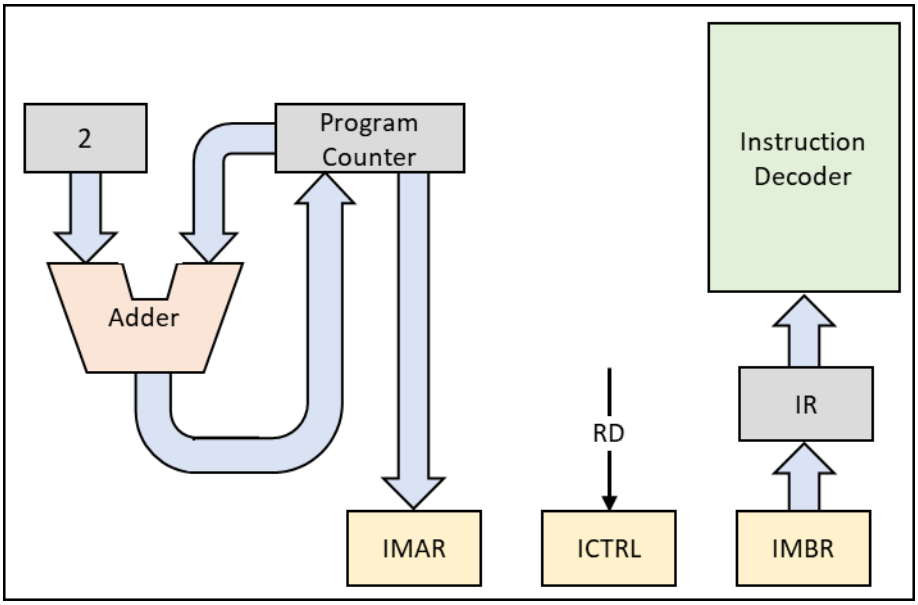

Figure 5: Instruction memory address register (IMAR) and buffer register (IMBR)

4 Register (CPU-resident) instructions

Register instructions are those that perform arithmetic, logic, move, and other non-memory access operations on registers. For example, adding R1 and R2:

```
ADD   R1,R2 ; R2 <- R2 + R1
```

still undergoes a Fetch, Decode, and Execute, but would be completed in two clock cycles, as shown in Figure 6.

Clock	Instruction	Stages			Comments
		Fetch	Decode	Execute	
N+0	`ADD R1,R2`	F0	D0		
N+1		F1		E0	End of previous instruction.
N+2	`MOV R2,R0`	F0	D0		
N+3		F1		E0	R2 ← R1 + R2

Figure 6: Example of pipelining a register instruction

Registers R1 and R2 are in the CPU's register file. R1 is selected by the control unit and placed on the S-bus and R2 is selected and placed on the D-bus (see Figure 7).

The ALU function is "ADD", causing the values on the S- and D-buses to be read by the ALU and the arithmetic to be performed.

When the ALU has finished the addition, the status of the operation is returned to the control unit. This is the Program Status Word (PSW). The PSW contains four status indicators: Zero, the result of the operation is zero; Negative, the result of the operation is negative (that is, the most significant bit is zero); Carry, the result of an unsigned arithmetic operation exceeds the word size (8- or 16-bits); and Overflow, the result of a signed arithmetic operation has resulted in an invalid sign.

The result is on the ALU bus and then copied to the destination register, R2, in the register file.

The Execute stage is ready for the next instruction.

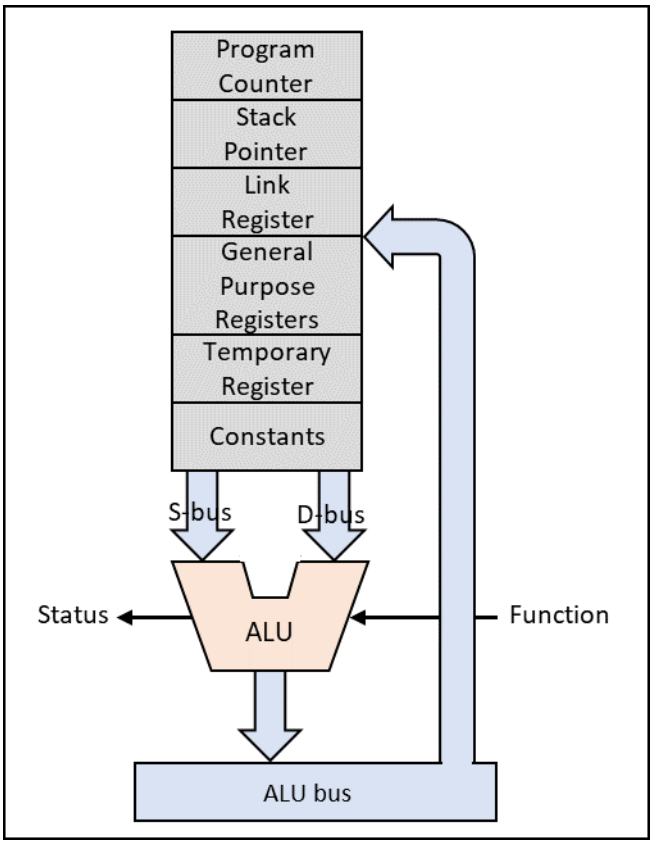

Figure 7: The Fetch stage internals

At the end of the execute stage (E0), any registers it has updated are available for use with the next instruction. We see this in Figure 8, where the end of the execution of one instruction allows the next instruction to use the same register if necessary.

Clock	Instruction	Stages			Comments
		Fetch	Decode	Execute	
N+0	`ADD R1,R2`	F0	D0		
N+1		F1		E0	End of previous instruction.
N+2	`MOV R2,R0`	F0	D0		
N+3		F1		E0	R2 ← R1 + R2
N+4	`AND #8,R2`	F0	D0		
N+5		F1		E0	R0 ← R2
N+6	`BIT #0,R2`	F0	D0		
N+7		F1		E0	R2 ← R2 & #8

Figure 8: Example of pipelining multiple register instructions

5 Memory Access (Load and Store)

One of the bottlenecks of the traditional von Neumann architecture is its inability to a load or a store to take place at the same time as an instruction fetch. This is the result of having a single memory for both instructions and data.

Depending on the pipeline's design, loads and stores might interfere with an instruction fetch. XM3p's design assumes that data and instructions are stored in separate memories. This, of course, is referred to as the *Harvard architecture*.

The XM-23p's data memory is accessed using three registers:

DMAR: Data Memory Address Register – holds the address of the memory to be accessed for either loading or storing. Write only.

DMBR: Data Memory Buffer Register – contains the value of the register to be written to memory or the contents of the most recently read memory location. Read or write. Word or byte. (Alternatively, this can be referred to as the Data Memory Data Register or DMDR.)

DCTRL: Data Memory Control register – indicates whether the location specified in the DMAR is to be read from or written to.

The ALU is used for register-register or register-constant arithmetic and logic operations. It is also used with the Load and Store instructions for indexed addressing and offset addressing.

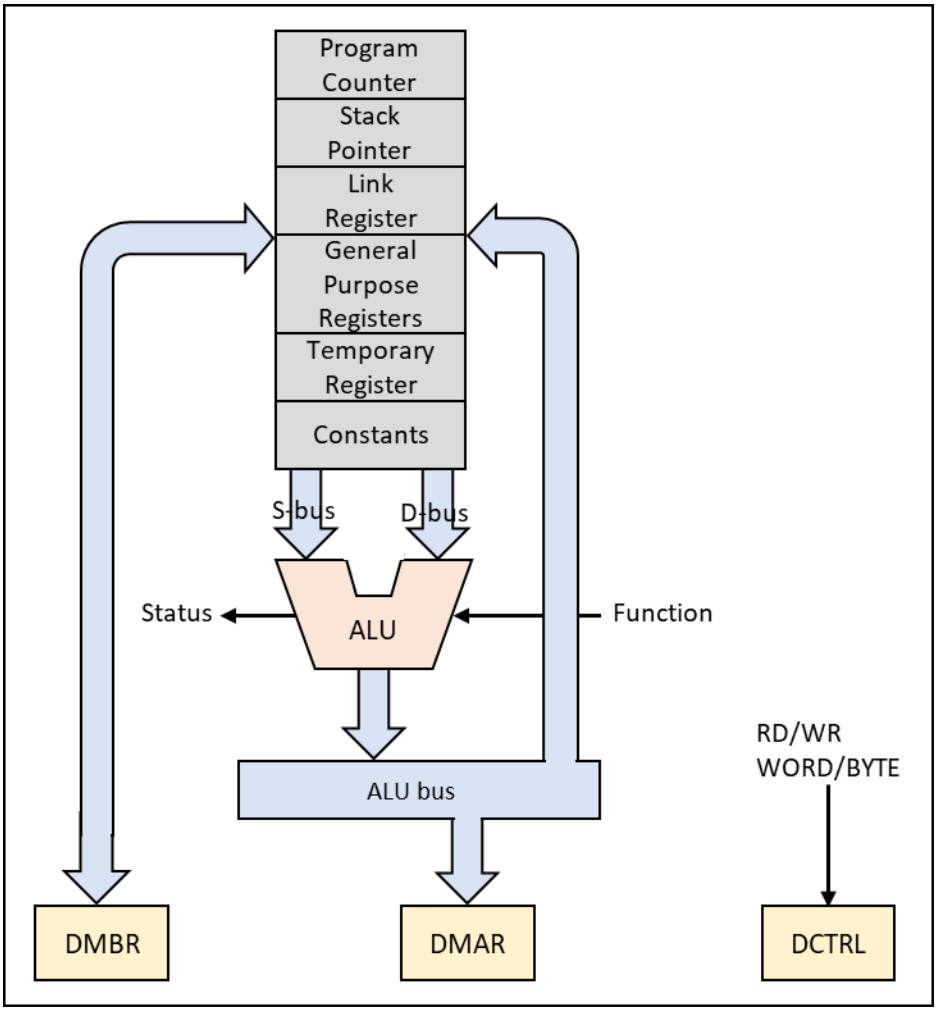

Figure 9: Data memory access

Accessing memory (load or store) occurs in the Execute stage and takes two clock cycles rather than one (see Figure 13). This has several consequences:

- The load and store instructions require an additional step (E1). This step occurs at the same time as the next F0 and D0.
- The instructions require three clock cycles between instructions rather than the usual two taken with other instructions.

We can represent the memory accessing as a state diagram (see Figure 10). States START and *Wait Tick 0* and their transitions are the same as those found in Figure 4.

State *Wait tick 1* now has two exit transitions. When the clock ticks, the current instruction is either performing a memory access or it is not. If it is performing a memory access (Mem Access is TRUE), steps E1, F0, and D0 are started, and the next state is *Wait Tick 2*. However, if there is not a memory access (Mem Access is FALSE), steps F0 and D0 are started, and the next state is *Wait tick 0*.

In state *Wait tick 2* the memory access has completed, and memory has either been written or read. At the next clock tick, steps F1 and E0 are started; the next state is *Wait tick 1*.

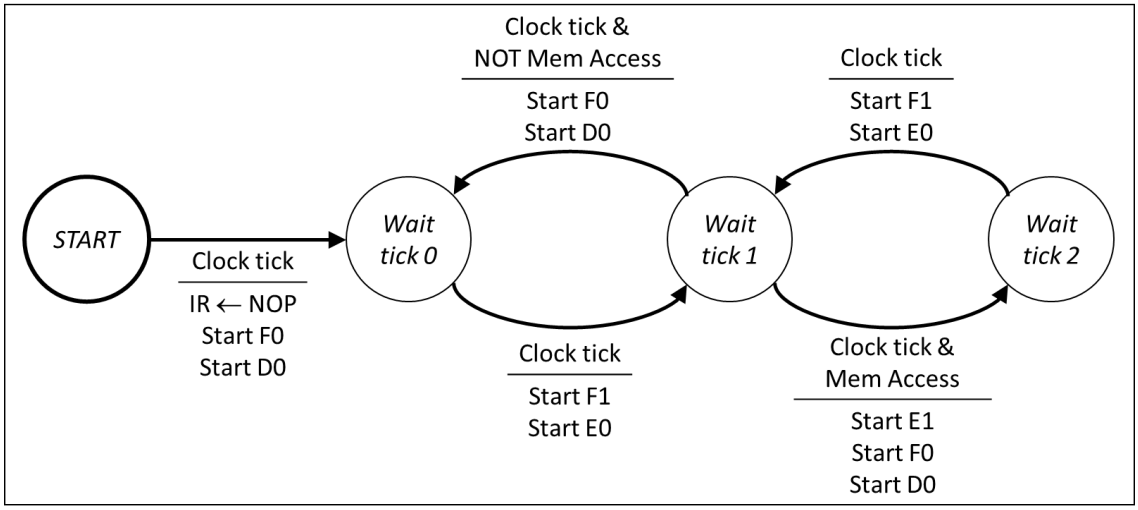

Figure 10: XM-23p state diagram with stage E1.

5.1 Load

Load instructions read the contents of a location in the machine's data memory into the destination register:

```
LD    R1,R2 ; R2 <- DMEM[R1]
ADD   #1,R1 ; R1 <- R1 + #1
```

The instruction takes five steps (F0, F1, D0, E0, and E1), but due to the overlapping of instructions in the pipeline, the instruction is completed in three clock cycles, as shown in Figure 11.

Clock	Instruction	Stages			Comments
		Fetch	Decode	Execute	
N+0	`LD R1,R2`	F0	D0		
N+1		F1		E0	End of previous instruction
N+2	`ADD #1,R1`	F0	D0		
N+3		F1		E0	DMAR ← R1; DCTRL ← RD
N+4	`MOV R2,R0`	F0	D0	E1	DMEM writes to DMBR; R2 ← DMBR
N+5		F1		E0	R1 ← R1 + #1; Update PSW

Figure 11: The load instruction stages and steps

The fetch and decode stages are the same as those used for all other instructions. They differ in the execution stage with two steps:

Step	Action
E0	• If the addressing mode is a pre- increment or decrement, or a relative offset, the addressing register is updated. • The addressing register is written to the Data Memory Address Register (DMAR). • READ is written to the control register (DCTRL) along with the size (WORD or BYTE). • If the addressing mode is post- increment or decrement, the addressing register is updated.
E1	• When memory is read, the contents of the location specified in the DMAR are read, the value is returned in the Data Memory Data Register (DMDR). • The DMDR is written to the destination register.

5.2 Store

The store instruction writes the contents of a register to a location in the machine's data memory, for example:

```
ST    R1,R2       ; DMEM[R2] <- R1
MOVL  #FF,R0      ; R0 <- #FF
```

The store instruction writes the contents of the destination register to the address specified in the source register (see Figure 12).

Clock	Instruction	Stages			Comments
		Fetch	Decode	Execute	
N+0	`ST R1,R2`	F0	D0		
N+1		F1		E0	End of previous instruction
N+2	`MOVL #FF,R0`	F0	D0		
N+3		F1		E0	DMBR ← R1; DMAR ← R2; DCTRL ← WR
N+4	`MOVH #1200,R2`	F0	D0	E1	Write MDBR to MDAR in DMEM

Figure 12: The store instruction stages and steps

The Execute stage steps of the store instruction are like those with the load instruction:

Step	Action
E0	• If the addressing mode is a pre- increment or decrement, or a relative offset, the addressing register is updated. • The addressing register is written to the Data Memory Address Register (DMAR). • The register to be stored is written to the DMDR.
E1	• WRITE is written to the control register (DCTRL) along with the size (WORD or BYTE). • If the addressing mode is post- increment or decrement, the addressing register is updated.

5.3 Addressing

XM-23p has three addressing modes: direct, relative, and indexed.

In direct-memory addressing, the value of the addressing register is written directly to the DMAR.

Relative addressing instructions add the offset to the index register in step E0. The updated address is written to the DMAR. The value of the register (i.e., the base address) is not changed.

Indexed addressing instructions can increment or decrement the addressing register before or after the memory access has taken place. If a pre-increment or pre-decrement is required, the addressing register is updated and then the new value of the register is written to the DMAR and the register file in step E0. In the case of a post-increment or a post-decrement, the value of the indexing register is written to the DMAR in step E0. The incrementing or decrementing operation is performed in step E1, before the load or store completes.

5.4 Data hazards

During a load, the destination register is not updated until step E1 is completed. This could be a *data hazard* if either steps F0 or D0 require the same register. Consider the following (used in Figure 13):

```
LD    R1,R2       ; R2 <– DMEM[R1]
ADD   #1,R2       ; R2 < R2 + #1
```

In a non-pipelined architecture, since each instruction is completed before the next fetch stage begins, R2 is available for the addition.

However, in a pipelined architecture, if R2 is required in the Fetch stage (as it could be in a memory access instruction) or in the Decode stage (preparing the control and data information for the Execute stage at the next clock cycle); for example, the second instruction (ADD #1,R2) might attempt to read R2 before the first instruction (LD R1,R2) has finished executing.

At time N+4 in Figure 13, if the decode of ADD #1,R2 (step D0) accesses R2 from the register file before the completion of LD R1,R2 (step E1) updates R2, the value of R2 will be incorrect when it is used when ADD #1,R2 is executed in step E0 at time N+5.

The order of events should be step D0 occurs after step E1 has executed. That is, the register should have been *read* from the register file *after* it was *written* to the register file. This is referred to as a **RAW** or a read-after-write data hazard.

Clock	Instruction	Stages			Comments
		Fetch	Decode	Execute	
N+0	`LD R1,R2`	F0	D0		
N+1		F1		E0	End of previous instruction
N+2	`ADD #1,R2`	F0	D0		
N+3		F1		E0	DMAR ← R1; DCTRL ← RD
N+4	`MOV R2,R0`	F0	D0	E1	DMEM writes to DMBR; R2 ← DMBR
N+5		F1		E0	R2 ← R2 + #1; Update PSW

Figure 13: Accessing data memory with potential data hazard

A data hazard, also referred to as *data dependency*, is equivalent to a *race condition*. The RAW data hazard is known as a *true dependency*. If a register should have been *written* to the register file *after* it was *read*, a **WAR** or write-after-read data hazard is said to have occurred; the WAR data hazard is not (or should not be) possible in XM-23p.

6 Branching

Branching instructions change the machine's flow of control by adding an offset to the value of the program counter (PC). On the next fetch, the PC refers to the location to be read.

6.1 Unconditional branches

In an unconditional branch, control passes to the address specified by the branch's address. Consider the following example:

```
Loop  ADD   #1,R0
      LD    R0,R3
      ...
      BR    Loop
Done  MOV   R2,R0
```

In the example, control returns to `Loop` when the branch is executed. Since `BR` is the branch instruction, control will never pass to label `Done` because the instruction after `BR` is executed is the `ADD` at `Loop`.

In a traditional von Neumann CISC machine, this is not an issue; however, in a pipeline processor it is an issue because the Fetch stage continues to execute.

We see this problem in Figure 14, where the instruction at label Done (`MOV R2,R0`) is fetched while the branch continues to execute. At time N+3, the Fetch stage (of the `MOV` instruction) has completed, making the instruction available for decoding; also at this time, the PC has been updated for the correct fetch at label `Loop` (`ADD #1,R0`).

If the `MOV R2,R0` instruction is allowed to continue, the machine will be put into an undefined state leading to unpredictable results. This is a *control* (or *branch*) *hazard*, and like the data hazard, must be avoided.

The commonly adopted solution is to insert a *bubble* into the instruction stream to avoid letting the instruction (`MOV R2,R0` in this case) entering the Decode and Execute stages. The bubble is typically a NOP instruction. Alternatively, steps D0 and E0 could be signalled to "do nothing" for their next steps.

Clock	Instruction	Stages			Comments
		Fetch	Decode	Execute	
N+0	`BR Loop`	F0	D0		
N+1		F1		E0	
N+2	`Done MOV R2,R0`	F0	D0		
N+3		F1		E0	PC ← PC + Offset
N+4	`Loop ADD #1,R0`	F0	~~D0~~		
N+5		F1		~~E0~~	~~R0 ← R2~~
N+6	`LD R0,R3`	F0	D0		
N+7		F1		E0	R0 ← R0 + 1; Update PSW

Figure 14: Unconditional branching and control hazards

With the PC changed to the address of `Loop`, the next instruction to be fetched is `ADD #1,R0`.

6.2 Conditional branches

In a conditional branch, there are two possible outcomes, either control passes to the address specified in the branch or control resumes at the next instruction. In the following sample code, the branch-if-equal passes to control to `Loop` if the PSW indicates that the zero bit is set (Z=1). If not, control passes to `Done`.

```
Loop  ADD   #1,R0
      LD    R0,R3
      ...
      BEQ   Loop
Done  MOV   R2,R0
      SVC   #0
```

There are two possible outcomes, either the branch is taken (Z=1, zero has been found)) or not taken (Z=0, zero has not been found).

If the condition is true, the branch is taken, returning control to Loop. The next instruction to execute should be `ADD #1,R0`; however, because of the time taken to execute the `BEQ` instruction, the instruction at label `Done`, the instruction `MOV R2,R0`, is executed. This is also a control hazard and must be avoided by inserting a bubble into the instruction stream.

Clock	Instruction	Stages			Comments
		Fetch	Decode	Execute	
N+0	`BEQ Loop`	F0	D0		
N+1		F1		E0	
N+2	`Done MOV R2,R0`	F0	D0		
N+3		F1		E0	IF PSW.Z = 1 THEN PC ← PC + offset
N+4	`Loop ADD #1,R0`	F0	~~D0~~		
N+5		F1		~~E0~~	~~R0 ← R2~~
N+6	`LD R0,R3`	F0	D0		
N+7		F1		E0	R0 ← R0 + 1; Update PSW

Figure 15: Actions if conditional branch is TRUE

However, if the condition is false, control does not pass to the label, `Loop`, instead the next instruction, `MOV R2,R0` is allowed to execute.

Clock	Instruction	Stages			Comments
		Fetch	Decode	Execute	
N+0	`BEQ Loop`	F0	D0		
N+1		F1		E0	
N+2	`MOV R2,R0`	F0	D0		
N+3		F1		E0	IF PSW.Z = 1 THEN PC ← PC + offset
N+4	`SVC #0`	F0	D0		
N+5		F1		E0	R0 ← R2
N+6	`SUB #3,R4`	F0	D0		
N+7		F1		E0	Make supervisor call

Figure 16: Actions if conditional branch is FALSE

7 The XM-23p CPU

The XM-23p CPU consists of the three distinct stages all governed by a controller, the Control Unit (see Figure 17):

- The Fetch stage adds two to the program counter when instructed by the controller; the new PC value is copied to the IMAR and updates the program counter. The ICTRL receives a read indication, which signals instruction memory to read a copy of the instruction in location IMAR (F0). The value is written back to the IMBR (F1). The IMBR is copied into the IR.
- At the start of cycle N+2, the Decode stage reads the IR and decodes the instruction into its opcode (operation code) and operands. This information is made available to the controller so that it can generate the correct sequence of events for the Execute stage.

  At N+2, the Fetch stage sets the instruction memory registers to access the next instruction.
- The Execution stage starts at cycle N+3 and performs the steps as required by the instruction. Memory load and store instructions will use the DMBR, DMAR, and DCTRL registers.

  If the instruction is a memory access (load or store), an additional Execution step, E1, is required. It overlaps with the start of the next Fetch state.

  Also, at cycle N+3, Fetch step F1 is occurring, reading the value of the next instruction to be executed.

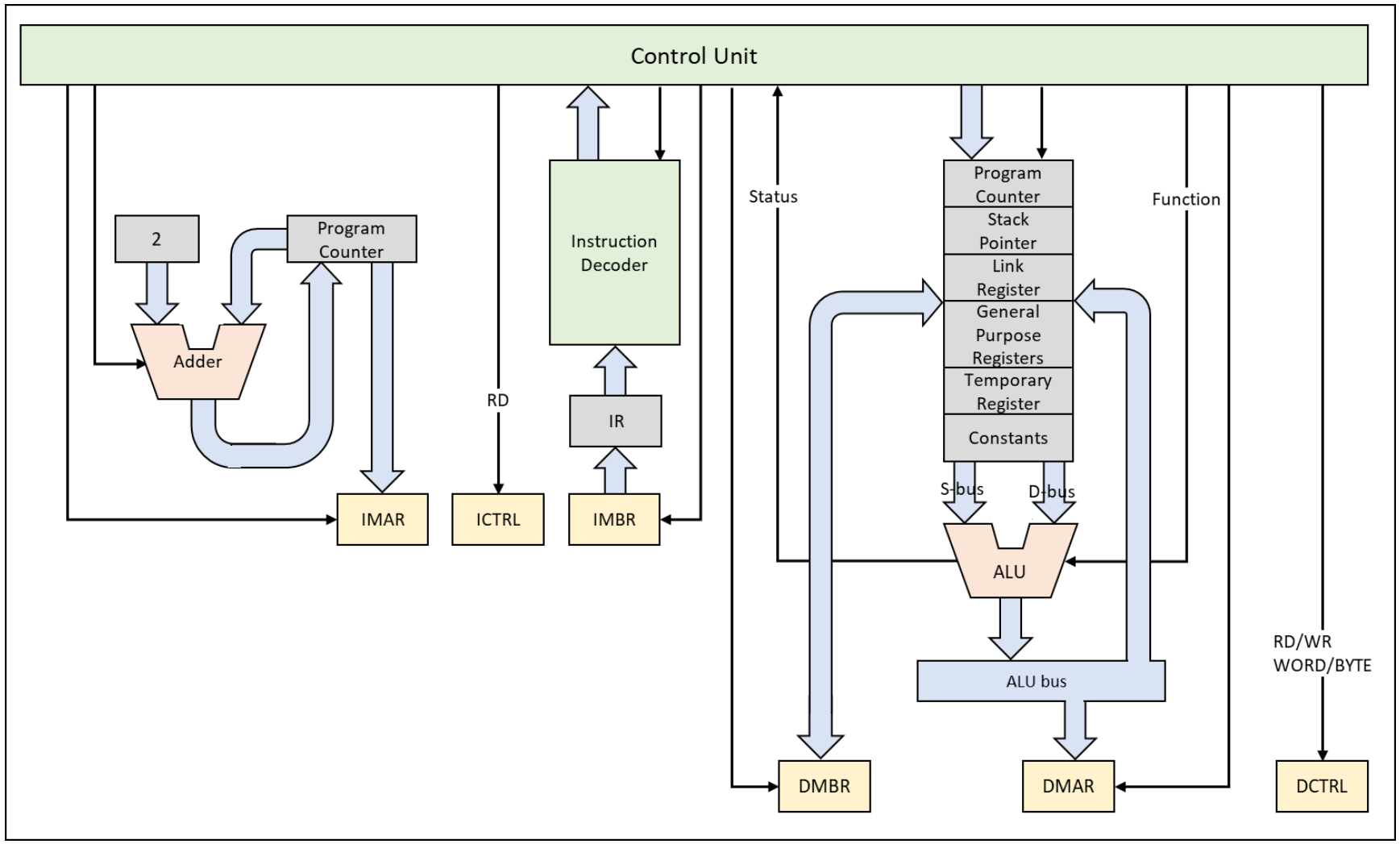

Figure 17: The complete XM-23p CPU
Note: The Program Counter is shared between the Fetch and the Execute stage