CS201 W'00 Lecture Notes
Lecture 9

Mic-1 Microcode Example

```
Main1     PC = PC + 1; fetch; goto (MBR)

pop1      MAR = SP = SP - 1; rd
pop2         // delay for read
pop3      TOS = MDR; goto Main1

goto1     OPC = PC - 1
goto2     PC = PC + 1; fetch
goto3     H = MBR << 8
goto4     H = MBRU OR H
goto5/6   PC = OPC + H; fetch; goto Main1
            // no delay needed

ifeq1     MAR = SP = SP - 1; rd
ifeq2     OPC = TOS
ifeq3     TOS = MDR
ifeq4     Z = OPC; if (Z) goto T; else goto F

T         OPC = PC - 1; goto goto2
            // no fetch needed

F         PC = PC + 1
F2/3      PC = PC + 1; fetch; goto Main1
            // no delay needed
```

From Mic-1 to Mic-2

Mic-1 uses very little hardware, but its cycles-per-instruction is high, and can be reduced.

Most important idea: mve responsibility for routine instruction and operand fetching into a separate **instruction fetch unit (IFU)**.

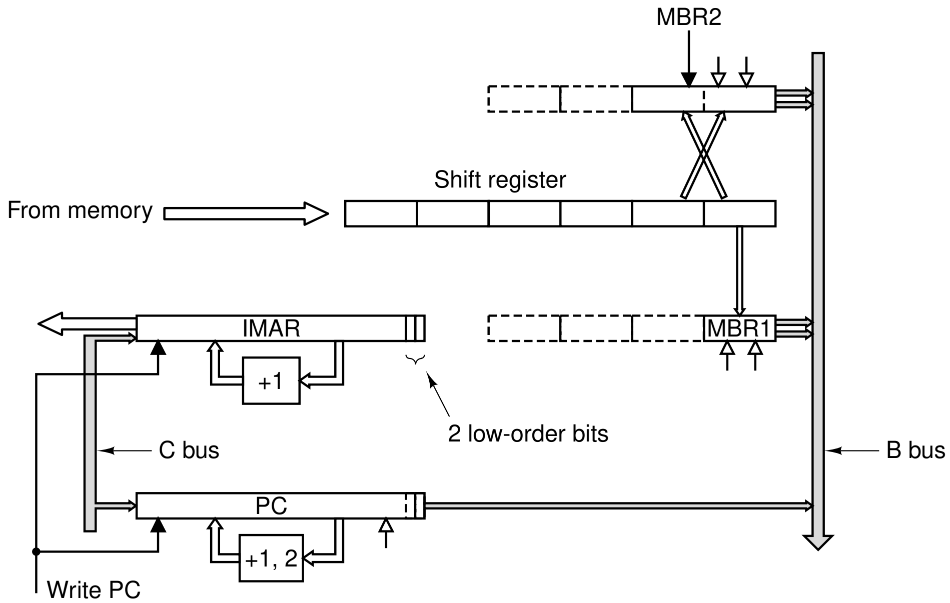

Figure 4-27. A fetch unit for the Mic-1.

Automatically **prefetches** instruction and operand bytes into `MBR1` and `MBR2`.

- Handles routine incrementing of PC, and watches for explicit writes to PC from main datapath.

- Precise memory timing model is not provided, but IFU clearly can stay ahead of datapath except (perhaps) after a branch is taken.

Mic-2 Architecture

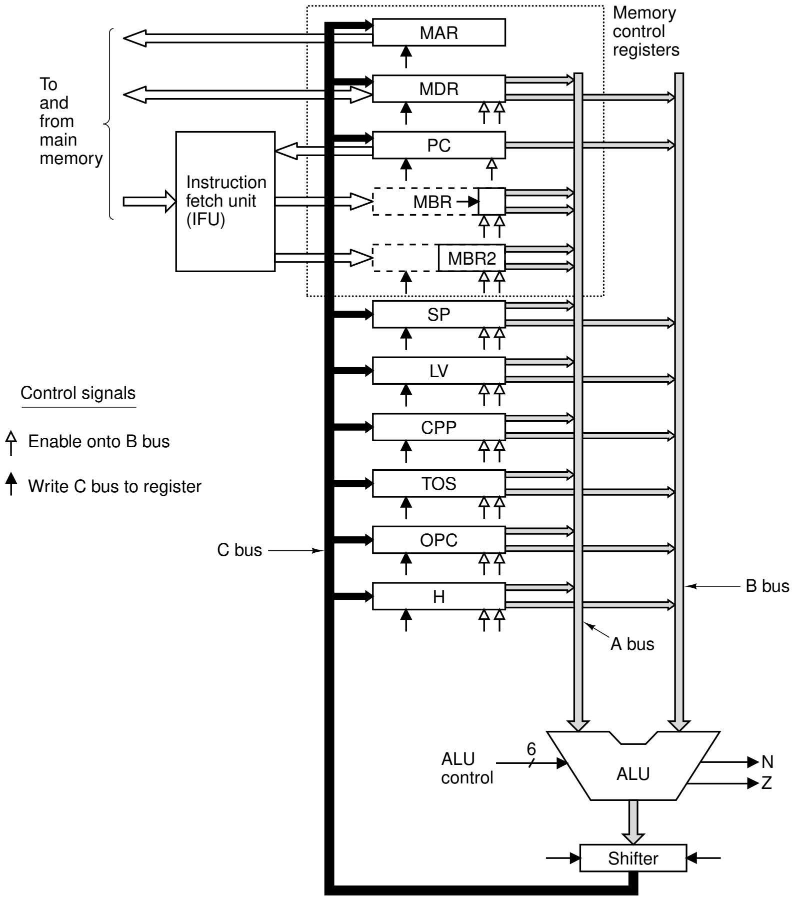

Figure 4-29. The datapath for Mic-2.

Also:

• Merge `Main1` into tail of each instruction where there is spare space. (No change to architecture required.)

• Add a separate A bus to allow any register to be used as left operand of ALU operation. (How many bits does this add to each microcode instruction?)

Mic-2 reduces cycle count for many instructions; overall gain depends on instruction frequency distribution.

Pipelining

Mic-2 remains highly **sequential**: for each microinstruction, it decodes the instruction, **then** reads register values, **then** does the ALU computation, **then** writes back results to registers.

Different part of the hardware does each phase of processing, and lies idle the rest of the time.

For a single microinstruction, can't really do better. But almost always have a known **sequence** of microinstructions to be executed in order.

Idea of pipelining: start working on early phases of next microinstruction while still doing later phases of previous one.

Although each microinstruction takes just as long to complete, overall IJVM instruction can complete much faster!

To make this work, must first divide existing clock cycle into smaller cycles, with intermediate results contained in **latches** (registers).

For example, if we latch register inputs and ALU result, we cut cycle time (roughly) by factor of three.

Of course, now need three cycles to complete the work of one original cycle. But can work on (different phases of) three different microinstructions at once!

Mic-3 Pipelined Architecture

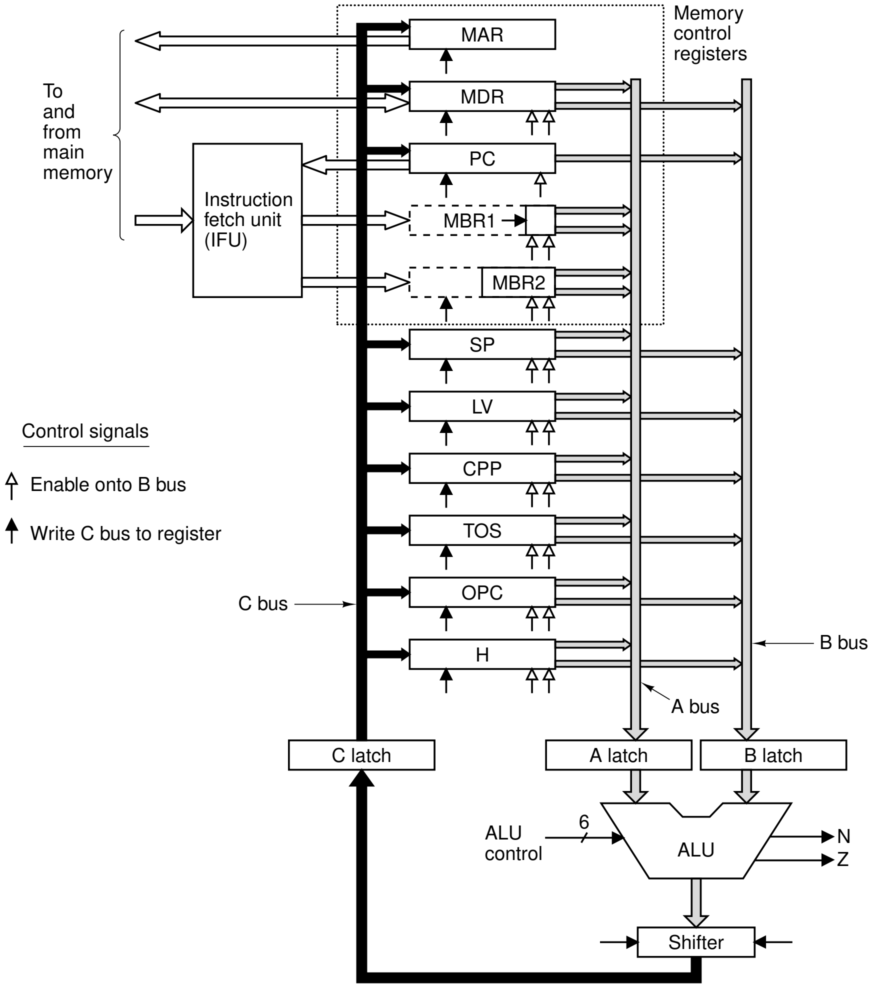

Figure 4-31. The three-bus data path used in the Mic-3.

Pipelining Microinstructions in Mic-3

	Swap1	Swap2	Swap3	Swap4	Swap5	Swap6
Cy	**MAR=SP−1;rd**	**MAR=SP**	**H=MDR;wr**	**MDR=TOS**	**MAR=SP−1;wr**	**TOS=H;goto (MBR1)**
1	B=SP					
2	C=B−1	B=SP				
3	MAR=C; rd	C=B				
4	MDR=mem	MAR=C				
5			B=MDR			
6			C=B	B=TOS		
7			H=C; wr	C=B	B=SP	
8			Mem=MDR	MDR=C	C=B−1	B=H
9					MAR=C; wr	C=B
10					Mem=MDR	TOS=C
11						goto (MBR1)

Figure 4-33. The implementation of SWAP on the Mic-3.

Pipelining Makes Parallel use of Hardware

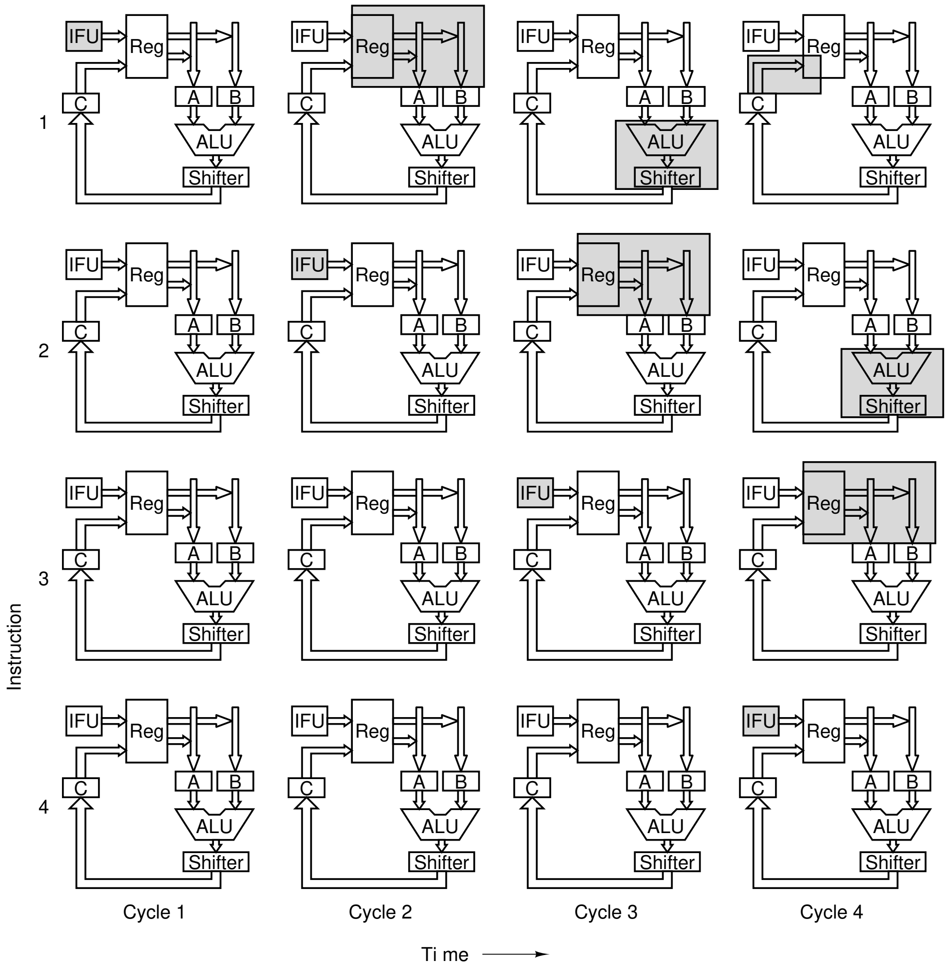

Figure 4-34. Graphical illustration of how a pipeline works.

Mic-4 Architecture

So far, pipelining only works for the microinstructions for a single IJVM instruction, since terminating `goto (MBR1)` completely stalls pipeline.

To fix this, need to separate IJVM instruction decoding into separate hardware that builds a **queue** of microinstructions (for more than single IJVM instruction). Can then pipeline processing of entire queue.

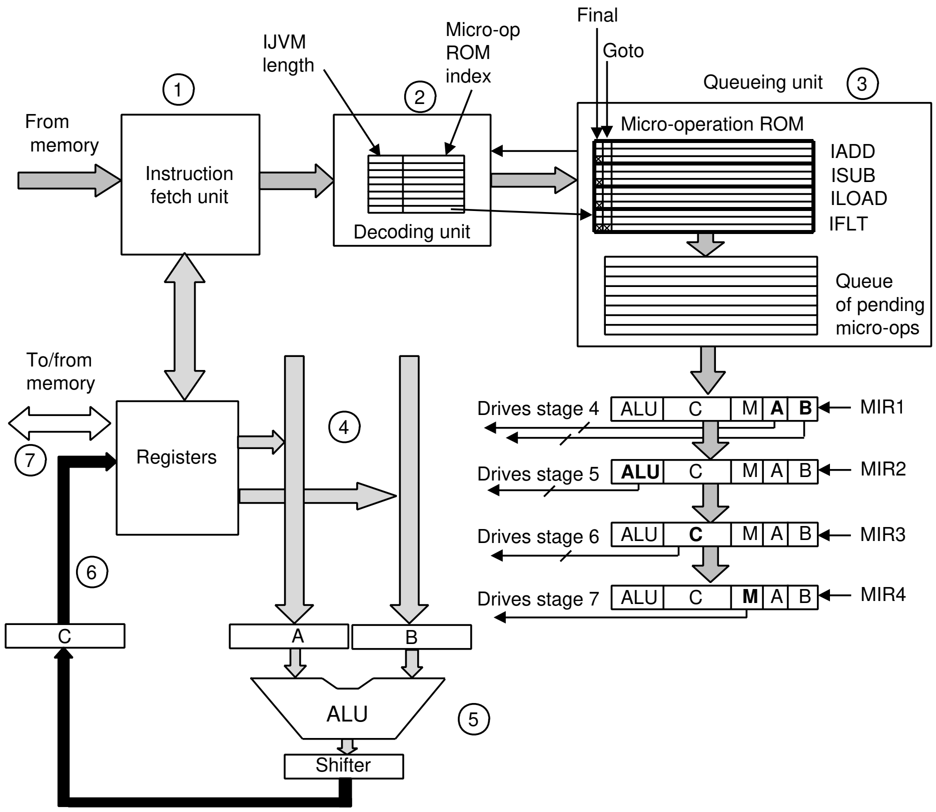

Figure 4-35. The main components of the Mic-4.

Get 7-stage pipeline. (Also shows microinstruction control more explicitly.)

Branch Prediction

Explicit branches in IVJM code still stall pipeline.

Even **unconditional** branches do so, because even though the target is known, we can't start fetching from it until the instruction has been decoded and the target address calculated, which is too late to avoid a stall.

(Hence the idea of **delay slots** on SPARC.)

Conditional branches are even worse, because we must also evaluate condition before knowing true next address.

To avoid excessive stalling, many machines try to **predict** whether conditional branches will be taken or not. They execute along predicted path, but back out if it turns out that the prediction was wrong. This gets very messy to implement!

Static prediction assigns a fixed guess to each branch. Example: backward branches are taken; forward branches are not. Newer architectures allow programmer (or compiler) to make the guess, by providing two forms of branch instruction (probably taken/probably not taken).

Dynamic prediction keeps a hardware history of which branches are taken and which are not, on the assumption that future branch behavior will be similar to past behavior. History may be one, two, or more bits.

Decent prediction is quite important to obtain benefits from pipelining!

Comparison of Real Architectures

Pentium II is similar to Mic-4: it breaks down its CISC instruction set into 3-register-operand microinstructions that are queued for pipelined execution.

UltraSPARC II already has simple 3-register-operand instructions that can be executed directly by a pipelined datapath, one per cycle.

Both these machines are **superscalar**: they have multiple ALU's and floating point units, permitting several arithmetic operations to be executed in parallel.

picoJava II executes JVM's memory-stack instructions, but:

• Top 64 words of stack are cached in registers (like TOS on Mic-n) with special hardware to keep them current.

• Some sequences of JVM instructions can be **folded** together to produce a single 3-register-operand microinstruction; others correspond directly to a single microinstruction; a few require multiple microinstructions.

All three machines have deep pipelines and use branch prediction to keep the pipelines filled.

Thus, the computational cores of all three machines are similar, despite large differences in their ISA's.