Design of Digital Circuits
Lecture 15: Pipelining

Prof. Onur Mutlu
ETH Zurich
Spring 2017
13 April 2017
Agenda for Today & Next Few Lectures

- Single-cycle Microarchitectures
- Multi-cycle and Microprogrammed Microarchitectures
- Pipelining
  - Issues in Pipelining: Control & Data Dependence Handling, State Maintenance and Recovery, ...
- Out-of-Order Execution
  - Issues in OoO Execution: Load-Store Handling, ...
Readings for This Week

- H&H, Chapter 7.5 (keep reading)
Wrap Up Microprogramming
Remember: An Exercise in Microprogramming
Handouts

- 7 pages of Microprogrammed LC-3b design

A Simple LC-3b Control and Datapath

Figure C.1: Microarchitecture of the LC-3b, major components
A Simple Datapath Can Become Very Powerful
APPENDIX C. THE MICR OARCHITECTURE OF THE LC-3B, B ASIC MACHINE

Address of Next State


IR[15:12]

State Machine for LDW

State 18 (010010)
State 33 (100001)
State 35 (100011)
State 32 (100000)
State 6 (000110)
State 25 (011001)
State 27 (011011)

Microsequencer
Figure C.6: Additional logic required to provide control signals LC-3b to operate correctly with a memory that takes multiple clock cycles to read or store a value.

Suppose it takes memory five cycles to read a value. That is, once MAR contains the address to be read and the microinstruction asserts READ, it will take five cycles before the contents of the specified location in memory are available to be loaded into MDR. (Note that the microinstruction asserts READ by means of three control signals: MIO.EN/YES, R.W/RD, and D.A.TA.SIZE/WORD; see Figure C.3.)

Recall our discussion in Section C.2 of the function of state 33, which accesses an instruction from memory during the fetch phase of each instruction cycle. For the LC-3b to operate correctly, state 33 must execute five times before moving on to state 35. That is, until MDR contains valid data from the memory location specified by the contents of MAR, we want state 33 to continue to re-execute. After five clock cycles, the memory has completed the "read," resulting in valid data in MDR, so the processor can move on to state 35.

What if the microarchitecture did not wait for the memory to complete the read operation before moving on to state 35? Since the contents of MDR would still be garbage, the microarchitecture would put garbage into IR in state 35. The ready signal (R) enables the memory read to execute correctly. Since the memory knows it needs five clock cycles to complete the read, it asserts a ready signal (R) throughout the fifth clock cycle. Figure C.2 shows that the next state is 33 (i.e., 100001) if the memory read will not complete in the current clock cycle and state 35 (i.e., 100011) if it will. As we have seen, it is the job of the microsequencer (Figure C.5) to produce the next state address.
<table>
<thead>
<tr>
<th>Signal Name</th>
<th>Signal Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD.MAR/1</td>
<td>NO, LOAD</td>
</tr>
<tr>
<td>LD.MDR/1</td>
<td>NO, LOAD</td>
</tr>
<tr>
<td>LD.IR/1</td>
<td>NO, LOAD</td>
</tr>
<tr>
<td>LD.BEN/1</td>
<td>NO, LOAD</td>
</tr>
<tr>
<td>LD.REG/1</td>
<td>NO, LOAD</td>
</tr>
<tr>
<td>LD.CC/1</td>
<td>NO, LOAD</td>
</tr>
<tr>
<td>LD.PC/1</td>
<td>NO, LOAD</td>
</tr>
<tr>
<td>GatePC/1</td>
<td>NO, YES</td>
</tr>
<tr>
<td>GateMDR/1</td>
<td>NO, YES</td>
</tr>
<tr>
<td>GateALU/1</td>
<td>NO, YES</td>
</tr>
<tr>
<td>GateMARMUX/1</td>
<td>NO, YES</td>
</tr>
<tr>
<td>GateSHF/1</td>
<td>NO, YES</td>
</tr>
<tr>
<td>PCMUX/2</td>
<td>PC+2; select pc+2</td>
</tr>
<tr>
<td></td>
<td>BUS; select value from bus</td>
</tr>
<tr>
<td></td>
<td>ADDER; select output of address adder</td>
</tr>
<tr>
<td>DRMUX/1</td>
<td>11.9; destination IR[11:9]</td>
</tr>
<tr>
<td></td>
<td>R7; destination R7</td>
</tr>
<tr>
<td>SR1MUX/1</td>
<td>11.9; source IR[11:9]</td>
</tr>
<tr>
<td></td>
<td>8.6; source IR[8:6]</td>
</tr>
<tr>
<td>ADDR1MUX/1</td>
<td>PC, BaseR</td>
</tr>
<tr>
<td>ADDR2MUX/2</td>
<td>ZERO; select the value zero</td>
</tr>
<tr>
<td></td>
<td>offset6; select SEXT[IR[5:0]]</td>
</tr>
<tr>
<td></td>
<td>PCOffset9; select SEXT[IR[8:0]]</td>
</tr>
<tr>
<td></td>
<td>PCOffset11; select SEXT[IR[10:0]]</td>
</tr>
<tr>
<td>MARMUX/1</td>
<td>7.0; select LSHF(ZEXT[IR[7:0]], 11)</td>
</tr>
<tr>
<td></td>
<td>ADDER; select output of address adder</td>
</tr>
<tr>
<td>ALUK/2</td>
<td>ADD, AND, XOR, PASSA</td>
</tr>
<tr>
<td>MIO.EN/1</td>
<td>NO, YES</td>
</tr>
<tr>
<td>R.W/1</td>
<td>RD, WR</td>
</tr>
<tr>
<td>DATA.SSIZE/1</td>
<td>BYTE, WORD</td>
</tr>
<tr>
<td>LSHF1/1</td>
<td>NO, YES</td>
</tr>
</tbody>
</table>

Table C.1: Data path control signals
Figure C.4: The control structure of a microprogrammed implementation, overall block diagram on the LC-3b instruction being executed during the current instruction cycle. This state carries out the DECODE phase of the instruction cycle. If the IRD control signal in the microinstruction corresponding to state 32 is 1, the output MUX of the microsequencer (Figure C.5) will take its source from the six bits formed by 00 concatenated with the four opcode bits IR[15:12]. Since IR[15:12] specifies the opcode of the current LC-3b instruction being processed, the next address of the control store will be one of 16 addresses, corresponding to the 14 opcodes plus the two unused opcodes, IR[15:12] = 1010 and 1011. That is, each of the 16 next states is the first state to be carried out after the instruction has been decoded in state 32. For example, if the instruction being processed is ADD, the address of the next state is state 1, whose microinstruction is stored at location 000001. Recall that IR[15:12] for ADD is 0001.
Figure C.5: The microsequencer of the LC-3b base machine.

Unused opcodes, the microarchitecture would execute a sequence of microinstructions, starting at state 10 or state 11, depending on which illegal opcode was being decoded.

In both cases, the sequence of microinstructions would respond to the fact that an instruction with an illegal opcode had been fetched.

Several signals necessary to control the data path and the microsequencer are not among those listed in Tables C.1 and C.2. They are DR, SR1, BEN, and R. Figure C.6 shows the additional logic needed to generate DR, SR1, and BEN. The remaining signal, R, is a signal generated by the memory in order to allow...
| BUS | CON | 000000 (State 0) | 000001 (State 1) | 000010 (State 2) | 000011 (State 3) | 000100 (State 4) | 000101 (State 5) | 000110 (State 6) | 000111 (State 7) | 001000 (State 8) | 001001 (State 9) | 001010 (State 10) | 001011 (State 11) | 001100 (State 12) | 001101 (State 13) | 001110 (State 14) | 001111 (State 15) | 010000 (State 16) | 010001 (State 17) | 010010 (State 18) | 010011 (State 19) | 010100 (State 20) | 010101 (State 21) | 010110 (State 22) | 010111 (State 23) | 011000 (State 24) | 011001 (State 25) | 011010 (State 26) | 011011 (State 27) | 011100 (State 28) | 011101 (State 29) | 011110 (State 30) | 011111 (State 31) | 100000 (State 32) | 100001 (State 33) | 100010 (State 34) | 100011 (State 35) | 100100 (State 36) | 100101 (State 37) | 100110 (State 38) | 100111 (State 39) | 101000 (State 40) | 101001 (State 41) | 101010 (State 42) | 101011 (State 43) | 101100 (State 44) | 101101 (State 45) | 101110 (State 46) | 101111 (State 47) | 110000 (State 48) | 110001 (State 49) | 110010 (State 50) | 110011 (State 51) | 110100 (State 52) | 110101 (State 53) | 110110 (State 54) | 110111 (State 55) | 111000 (State 56) | 111001 (State 57) | 111010 (State 58) | 111011 (State 59) | 111100 (State 60) | 111101 (State 61) | 111110 (State 62) | 111111 (State 63) |
End of the Exercise in Microprogramming
Variable-Latency Memory

- The ready signal (R) enables memory read/write to execute correctly
  - Example: transition from state 33 to state 35 is controlled by the R bit asserted by memory when memory data is available

- Could we have done this in a single-cycle microarchitecture?

- What did we assume about memory and registers in a single-cycle microarchitecture?
The Microsequencer: Advanced Questions

- What happens if the machine is interrupted?
- What if an instruction generates an exception?
- How can you implement a complex instruction using this control structure?
  - Think REP MOVS instruction in x86
The Power of Abstraction

- The concept of a control store of microinstructions enables the hardware designer with a new abstraction: microprogramming.

- The designer can translate any desired operation to a sequence of microinstructions.

- All the designer needs to provide is:
  - The sequence of microinstructions needed to implement the desired operation.
  - The ability for the control logic to correctly sequence through the microinstructions.
  - Any additional datapath elements and control signals needed (no need if the operation can be “translated” into existing control signals).
Let’s Do Some More Microprogramming

- Implement REP MOVS in the LC-3b microarchitecture

- What changes, if any, do you make to the
  - state machine?
  - datapath?
  - control store?
  - microsequencer?

- Show all changes and microinstructions

- Extra Credit Assignment
REP MOVS (DEST SRC)

IF AddressSize = 16
    THEN
        Use CX for CountReg;
    ELSE IF AddressSize = 64 and REX.W used
        THEN Use RCX for CountReg; Fl;
    ELSE
        Use ECX for CountReg;
    Fl;
WHILE CountReg ≠ 0
    DO
        Service pending interrupts (if any);
        Execute associated string instruction;
        CountReg ← (CountReg - 1);
        IF CountReg = 0
            THEN exit WHILE loop; Fl;
        IF (Repeat prefix is REPZ or REPE) and (ZF = 0)
            or (Repeat prefix is REPNZ or REPNE) and (ZF = 1)
            THEN exit WHILE loop; Fl;
    OD;

How many instructions does this take in MIPS ISA?

How many microinstructions does this take to add to the LC-3b microarchitecture?
Aside: Alignment Correction in Memory

- **Unaligned accesses**

- LC-3b has byte load and byte store instructions that move data not aligned at the word-address boundary
  - Convenience to the programmer/compiler

- How does the hardware ensure this works correctly?
  - Take a look at state 29 for LDB
  - States 24 and 17 for STB
  - Additional logic to handle unaligned accesses

- P&P, Revised Appendix C.5
Aside: Memory Mapped I/O

- Address control logic determines whether the specified address of LDW and STW are to memory or I/O devices

- Correspondingly enables memory or I/O devices and sets up muxes

- An instance where the final control signals of some datapath elements (e.g., MEM.EN or INMUX/2) cannot be stored in the control store
  - These signals are dependent on memory address

- P&P, Revised Appendix C.6
Advantages of Microprogrammed Control

- Allows a very simple design to do powerful computation by controlling the datapath (using a sequencer)
  - High-level ISA translated into microcode (sequence of u-instructions)
  - Microcode (u-code) enables a minimal datapath to emulate an ISA
  - Microinstructions can be thought of as a user-invisible ISA (u-ISA)

- Enables easy extensibility of the ISA
  - Can support a new instruction by changing the microcode
  - Can support complex instructions as a sequence of simple microinstructions (e.g., REP MOVS, INC [MEM])

- Enables update of machine behavior
  - A buggy implementation of an instruction can be fixed by changing the microcode in the field
    - Easier if datapath provides ability to do the same thing in different ways
Update of Machine Behavior

- The ability to update/patch microcode in the field (after a processor is shipped) enables
  - Ability to add new instructions without changing the processor!
  - Ability to “fix” buggy hardware implementations

Examples

- IBM 370 Model 145: microcode stored in main memory, can be updated after a reboot
- IBM System z: Similar to 370/145.
- B1700 microcode can be updated while the processor is running
  - User-microprogrammable machine!
Multi-Cycle vs. Single-Cycle uArch

- Advantages
- Disadvantages
- For you to fill in
Can We Do Better?
Can We Do Better?

- What limitations do you see with the multi-cycle design?

- Limited concurrency
  - Some hardware resources are idle during different phases of instruction processing cycle
  - “Fetch” logic is idle when an instruction is being “decoded” or “executed”
  - Most of the datapath is idle when a memory access is happening
Can We Use the Idle Hardware to Improve Concurrency?

- **Goal:** More concurrency $\rightarrow$ Higher instruction throughput (i.e., more “work” completed in one cycle)

- **Idea:** When an instruction is using some resources in its processing phase, **process other instructions on idle resources** not needed by that instruction
  - E.g., when an instruction is being decoded, fetch the next instruction
  - E.g., when an instruction is being executed, decode another instruction
  - E.g., when an instruction is accessing data memory (ld/st), execute the next instruction
  - E.g., when an instruction is writing its result into the register file, access data memory for the next instruction
Pipelining
Pipelining: Basic Idea

- More systematically:
  - Pipeline the execution of multiple instructions
  - Analogy: “Assembly line processing” of instructions

- Idea:
  - Divide the instruction processing cycle into distinct “stages” of processing
  - Ensure there are enough hardware resources to process one instruction in each stage
  - Process a different instruction in each stage
    - Instructions consecutive in program order are processed in consecutive stages

- Benefit: Increases instruction processing throughput (1/CPI)
- Downside: Start thinking about this...
Example: Execution of Four Independent ADDs

- **Multi-cycle**: 4 cycles per instruction

- **Pipelined**: 4 cycles per 4 instructions (steady state)

Is life always this beautiful?
The Laundry Analogy

- "place one dirty load of clothes in the washer"
- "when the washer is finished, place the wet load in the dryer"
- "when the dryer is finished, take out the dry load and fold"
- "when folding is finished, ask your roommate (??) to put the clothes away"

- steps to do a load are sequentially dependent
- no dependence between different loads
- different steps do not share resources
Pipelining Multiple Loads of Laundry

- 4 loads of laundry in parallel
- no additional resources
- throughput increased by 4
- latency per load is the same

Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]
Pipelining Multiple Loads of Laundry: In Practice

Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]
Pipelining Multiple Loads of Laundry: In Practice

throughput restored (2 loads per hour) using 2 dryers
An Ideal Pipeline

- **Goal:** Increase throughput with little increase in cost (hardware cost, in case of instruction processing)

- Repetition of **identical operations**
  - The same operation is repeated on a large number of different inputs (e.g., all laundry loads go through the same steps)

- Repetition of **independent operations**
  - No dependencies between repeated operations

- **Uniformly partitionable suboperations**
  - Processing can be evenly divided into uniform-latency suboperations (that do not share resources)

- Fitting examples: automobile assembly line, doing laundry
  - What about the instruction processing “cycle”?
Ideal Pipelining

combinational logic (F,D,E,M,W)
T psec

BW = ~\(\frac{1}{T}\)

T/2 ps (F,D,E) → T/2 ps (M,W)

BW = ~\(\frac{2}{T}\)

T/3 ps (F,D) → T/3 ps (E,M) → T/3 ps (M,W)

BW = ~\(\frac{3}{T}\)
More Realistic Pipeline: Throughput

- Nonpipelined version with delay $T$
  
  $BW = \frac{1}{(T+S)}$ where $S =$ latch delay

- $k$-stage pipelined version
  
  $BW_{k\text{-stage}} = \frac{1}{(T/k + S)}$
  
  $BW_{\text{max}} = \frac{1}{(1 \text{ gate delay} + S)}$

Latch delay reduces throughput (switching overhead b/w stages)
More Realistic Pipeline: Cost

- Nonpipelined version with combinational cost $G$
  \[ \text{Cost} = G + L \text{ where } L = \text{latch cost} \]

- $k$-stage pipelined version
  \[ \text{Cost}_{k\text{-stage}} = G + Lk \]
  
  Latches increase hardware cost

Diagram:
- Nonpipelined version with $G$ gates and latch cost $L$.
- $k$-stage pipelined version with $G/k$ gates and latch cost $Lk$.
Pipelining Instruction Processing
Remember: The Instruction Processing Cycle

1. Instruction fetch (IF)
2. Instruction decode and register operand fetch (ID/RF)
3. Execute/Evaluate memory address (EX/AG)
4. Memory operand fetch (MEM)
5. Store/writeback result (WB)
Remember the Single-Cycle Uarch

Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]
Is this the correct partitioning? Why not 4 or 6 stages? Why not different boundaries?
5-stage speedup is 4, not 5 as predicted by the ideal model. Why?
Enabling Pipelined Processing: Pipeline Registers

No resource is used by more than 1 stage!
Pipelined Operation Example

All instruction classes must follow the same path and timing through the pipeline stages.

Any performance impact?

Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]
Pipelined Operation Example

Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]
Illustrating Pipeline Operation: Operation View

Inst_0

Inst_1

Inst_2

Inst_3

Inst_4

steady state
(full pipeline)
Illustrating Pipeline Operation: Resource View

<table>
<thead>
<tr>
<th></th>
<th>$t_0$</th>
<th>$t_1$</th>
<th>$t_2$</th>
<th>$t_3$</th>
<th>$t_4$</th>
<th>$t_5$</th>
<th>$t_6$</th>
<th>$t_7$</th>
<th>$t_8$</th>
<th>$t_9$</th>
<th>$t_{10}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>$l_0$</td>
<td>$l_1$</td>
<td>$l_2$</td>
<td>$l_3$</td>
<td>$l_4$</td>
<td>$l_5$</td>
<td>$l_6$</td>
<td>$l_7$</td>
<td>$l_8$</td>
<td>$l_9$</td>
<td>$l_{10}$</td>
</tr>
<tr>
<td>ID</td>
<td>$l_0$</td>
<td>$l_1$</td>
<td>$l_2$</td>
<td>$l_3$</td>
<td>$l_4$</td>
<td>$l_5$</td>
<td>$l_6$</td>
<td>$l_7$</td>
<td>$l_8$</td>
<td>$l_9$</td>
<td></td>
</tr>
<tr>
<td>EX</td>
<td>$l_0$</td>
<td>$l_1$</td>
<td>$l_2$</td>
<td>$l_3$</td>
<td>$l_4$</td>
<td>$l_5$</td>
<td>$l_6$</td>
<td>$l_7$</td>
<td>$l_8$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEM</td>
<td>$l_0$</td>
<td>$l_1$</td>
<td>$l_2$</td>
<td>$l_3$</td>
<td>$l_4$</td>
<td>$l_5$</td>
<td>$l_6$</td>
<td>$l_7$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WB</td>
<td>$l_0$</td>
<td>$l_1$</td>
<td>$l_2$</td>
<td>$l_3$</td>
<td>$l_4$</td>
<td>$l_5$</td>
<td>$l_6$</td>
<td>$l_7$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Control Points in a Pipeline

Identical set of control points as the single-cycle datapath!!
Control Signals in a Pipeline

- For a given instruction
  - same control signals as single-cycle, but
  - control signals required at different cycles, depending on stage

  ⇒ Option 1: decode once using the same logic as single-cycle and buffer signals until consumed

  ⇒ Option 2: carry relevant “instruction word/field” down the pipeline and decode locally within each or in a previous stage

Which one is better?
Pipelined Control Signals

Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]
Another Example: Single-Cycle and Pipelined
Another Example: Correct Pipelined Datapath

- **WriteReg must arrive at the same time as Result**
Another Example: Pipelined Control

- Same control unit as single-cycle processor
- Control delayed to proper pipeline stage
Remember: An Ideal Pipeline

- **Goal:** Increase throughput with little increase in cost (hardware cost, in case of instruction processing)

- Repetition of **identical operations**
  - The same operation is repeated on a large number of different inputs (e.g., all laundry loads go through the same steps)

- Repetition of **independent operations**
  - No dependencies between repeated operations

- **Uniformly partitionable suboperations**
  - Processing can be evenly divided into uniform-latency suboperations (that do not share resources)

- **Fitting examples:** automobile assembly line, doing laundry
  - What about the instruction processing “cycle”?
Instruction Pipeline: Not An Ideal Pipeline

- **Identical operations ... NOT!**
  ⇒ different instructions → not all need the same stages
  Forcing different instructions to go through the same pipe stages
  → external fragmentation (some pipe stages idle for some instructions)

- **Uniform suboperations ... NOT!**
  ⇒ different pipeline stages → not the same latency
  Need to force each stage to be controlled by the same clock
  → internal fragmentation (some pipe stages are too fast but all take the same clock cycle time)

- **Independent operations ... NOT!**
  ⇒ instructions are not independent of each other
  Need to detect and resolve inter-instruction dependencies to ensure the pipeline provides correct results
  → pipeline stalls (pipeline is not always moving)
Issues in Pipeline Design

- Balancing work in pipeline stages
  - How many stages and what is done in each stage

- Keeping the pipeline correct, moving, and full in the presence of events that disrupt pipeline flow
  - Handling dependences
    - Data
    - Control
  - Handling resource contention
  - Handling long-latency (multi-cycle) operations

- Handling exceptions, interrupts

- Advanced: Improving pipeline throughput
  - Minimizing stalls
Causes of Pipeline Stalls

- **Stall**: A condition when the pipeline stops moving

- **Resource contention**

- **Dependences** (between instructions)
  - Data
  - Control

- **Long-latency (multi-cycle) operations**
Dependences and Their Types

- Also called “dependency” or 
  less desirably “hazard”

- Dependences dictate ordering requirements between instructions

- Two types
  - Data dependence
  - Control dependence

- Resource contention is sometimes called resource dependence
  - However, this is not fundamental to (dictated by) program semantics, so we will treat it separately
Handling Resource Contention

- Happens when instructions in two pipeline stages need the same resource

- **Solution 1:** Eliminate the cause of contention
  - Duplicate the resource or increase its throughput
    - E.g., use separate instruction and data memories (caches)
    - E.g., use multiple ports for memory structures

- **Solution 2:** Detect the resource contention and stall one of the contending stages
  - Which stage do you stall?
  - Example: What if you had a single read and write port for the register file?
Example Resource Dependence: RegFile

- The register file can be read and written in the same cycle:
  - write takes place during the 1st half of the cycle
  - read takes place during the 2nd half of the cycle => no problem!!
  - However operations that involve register file have only *half a clock cycle* to complete the operation!!

```
<table>
<thead>
<tr>
<th>Time (cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

add $s0, $s2, $s3

and $t0, $s0, $s1

or $t1, $s4, $s0

sub $t2, $s0, $s5
```