4.6 Pipelined datapath and control

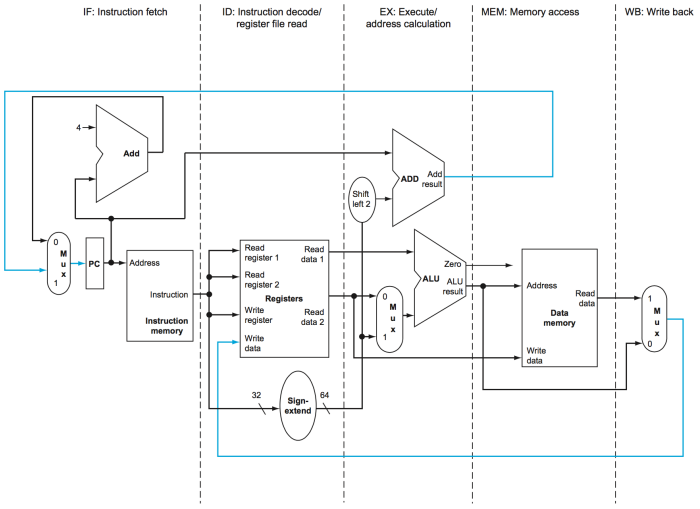
> "There is less in this than meets the eye."
> *Tallulah Bankhead, remark to Alexander Woollcott, 1922.*

The figure below shows the single-cycle datapath from COD Section 4.4 (A simple implementation scheme) with the pipeline stages identified. The division of an instruction into five stages means a five-stage pipeline, which in turn means that up to five instructions will be in execution during any single clock cycle. Thus, we must separate the datapath into five pieces, with each piece named corresponding to a stage of instruction execution:

1. IF: Instruction fetch
2. ID: Instruction decode and register file read
3. EX: Execution or address calculation
4. MEM: Data memory access
5. WB: Write back

Figure 4.6.1: The single-cycle datapath from COD Section 4.4 (A simple implementation scheme) (COD Figure 4.32).

Each step of the instruction can be mapped onto the datapath from left to right. The only exceptions are the update of the PC and the write-back step, shown in color, which sends either the ALU result or the data from memory to the left to be written into the register file. (Normally we use color lines for control, but these are data lines.)



In the figure above, these five components correspond roughly to the way the datapath is drawn; instructions and data move generally from left to right through the five stages as they complete execution. Returning to our laundry analogy, clothes get cleaner, drier, and more organized as they move through the line, and they never move backward.

There are, however, two exceptions to this left-to-right flow of instructions:

- The write-back stage, which places the result back into the register file in the middle of the datapath
- The selection of the next value of the PC, choosing between the incremented PC and the branch address from the MEM stage

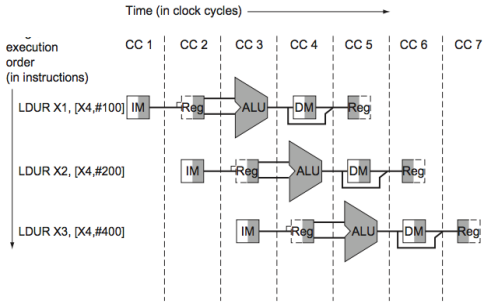
Data flowing from right to left does not affect the current instruction; these reverse data movements influence only later instructions in the pipeline. Note that the first right-to-left flow of data can lead to data hazards and the second leads to control hazards.

One way to show what happens in pipelined execution is to pretend that each instruction has its own datapath, and then to place these datapaths on a timeline to show their relationship. The figure below shows the execution of the instructions in COD Figure 4.26 (Single-cycle, nonpipelined execution (top) versus pipelined execution (bottom)) by displaying their private datapaths on a common timeline. We use a stylized version of the datapath in the figure above to show the relationships in the figure below.

Similar to previous figures, the animation below pretends that each instruction has its own datapath, and shades each portion according to use. Unlike those figures, each stage is labeled by the physical resource used in that stage, corresponding to the portions of the datapath in the figure above. *IM* represents the instruction memory and the PC in the instruction fetch stage, *Reg* stands for the register file and sign extender in the instruction decode/register file read stage (ID), and so on. To maintain proper time order, this stylized datapath breaks the register file into two logical parts: registers read during register fetch (ID) and registers written during write back (WB). This dual use is represented by drawing the unshaded left half of the register file using dashed lines in the ID stage, when it is not being written, and the unshaded right half in dashed lines in the WB stage, when it is not being read. As before, we assume the register file is written in the first half of the clock cycle and the register file is read during the second half.

PARTICIPATION ACTIVITY 4.6.1: Instructions being executed using the single-cycle datapath in the figure above, assuming pipelined execution (COD Figure 4.33).

Start ☐ 2x speed



PARTICIPATION ACTIVITY 4.6.2: Depicting pipeline stages.

Match the pipeline stage with the physical resource whose icon represents that stage in the stylized datapath depictions.

Reg (read) Reg (write) IM DM ALU

Stage 1: IF (instruction fetch)

Stage 2: ID (instruction decode)

Stage 3: EX (execute)

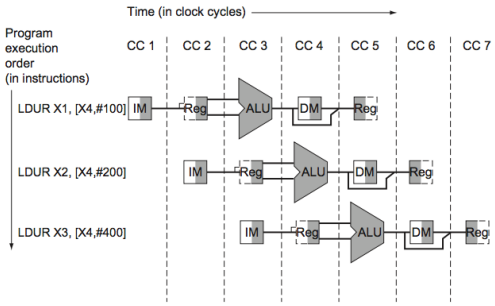
Stage 4: MEM (data memory access)

Stage 5: WB (write back)

Reset

PARTICIPATION ACTIVITY 4.6.3: Pipelined execution depiction for three LDUR instructions.

Consider the figure below showing pipelined execution of three instructions. Assume Instr1 is `LDUR X1, [X4,#100]`, Instr2 is `LDUR X2, [X4,#200]`, and Instr3 is `LDUR X3, [X4, #400]`. CC is short for "clock cycle".



1) In CC 1, Instr1 and Instr2 are executing.
   - ○ True
   - ○ False

2) In CC 2, Instr1 and Instr2 are both in stage IM.
   - ○ True
   - ○ False

3) The figure indicates that three ALUs are needed.
   - ○ True
   - ○ False

4) Two instructions could possibly need to use the ALU simultaneously.
   - ○ True
   - ○ False

5) Two instructions could possibly access the register file simultaneously.
   - ○ True
   - ○ False

6) Assuming a program has hundreds of instructions (rather than just three as shown above), how many instructions might possibly be executing during one clock cycle?

- ○ 3
- ○ 5

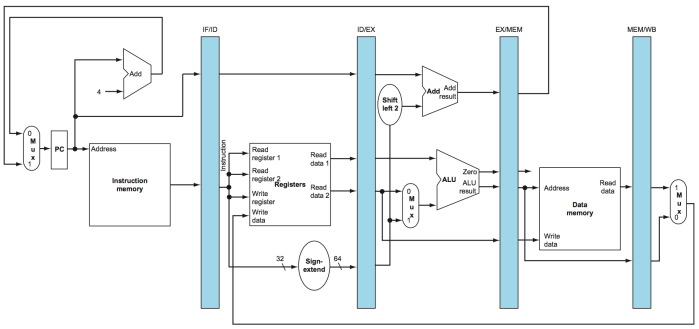
The animation above seems to suggest that three instructions need three datapaths. Instead, we add registers to hold data so that portions of a single datapath can be shared during instruction execution.

For example, as the animation above shows, the instruction memory is used during only one of the five stages of an instruction, allowing it to be shared by following instructions during the other four stages. To retain the value of an individual instruction for its other four stages, the value read from instruction memory must be saved in a register. Similar arguments apply to every pipeline stage, so we must place registers wherever there are dividing lines between stages in COD Figure 4.32 (The single-cycle datapath ...). Returning to our laundry analogy, we might have a basket between each pair of stages to hold the clothes for the next step.

The figure below shows the pipelined datapath with the pipeline registers highlighted. All instructions advance during each clock cycle from one pipeline register to the next. The registers are named for the two stages separated by that register. For example, the pipeline register between the IF and ID stages is called IF/ID.

Figure 4.6.2: The pipelined version of the single-cycle datapath (COD Figure 4.34).

The pipeline registers, in color, separate each pipeline stage. They are labeled by the stages that they separate; for example, the first is labeled *IF/ID* because it separates the instruction fetch and instruction decode stages. The registers must be wide enough to store all the data corresponding to the lines that go through them. For example, the *IF/ID* register must be 96 bits wide, because it must hold both the 32-bit instruction fetched from memory and the incremented 64-bit PC address. We will expand these registers over the course of this chapter, but for now the other three pipeline registers contain 256, 193, and 128 bits, respectively.



Notice that there is no pipeline register at the end of the write-back stage. All instructions must update some state in the processor—the register file, memory, or the PC—so a separate pipeline register is redundant to the state that is updated. For example, a load instruction will place its result in one of the 32 registers, and any later instruction that needs that data will simply read the appropriate register.

Of course, every instruction updates the PC, whether by incrementing it or by setting it to a branch destination address. The PC can be thought of as a pipeline register: one that feeds the IF stage of the pipeline. Unlike the shaded pipeline registers in the figure above, however, the PC is part of the visible architectural state; its contents must be saved when an exception occurs, while the contents of the pipeline registers can be discarded. In the laundry analogy, you could think of the PC as corresponding to the basket that holds the load of dirty clothes before the wash step.

To show how the pipelining works, throughout this chapter we show sequences of figures to demonstrate operation over time. These extra figures would seem to require much more time for you to understand. Fear not; the sequences take much less time than it might appear, because you can compare them to see what changes occur in each clock cycle. COD Section 4.7 (Data hazards: Forwarding versus stalling) describes what happens when there are data hazards between pipelined instructions; ignore them for now.

The next three figures, our first sequence, show the active portions of the datapath highlighted as a load instruction goes through the five stages of pipelined execution. We show a load first because it is active in all five stages. We highlight the *right half* of registers or memory when they are being *read* and highlight the *left half* when they are being *written*.

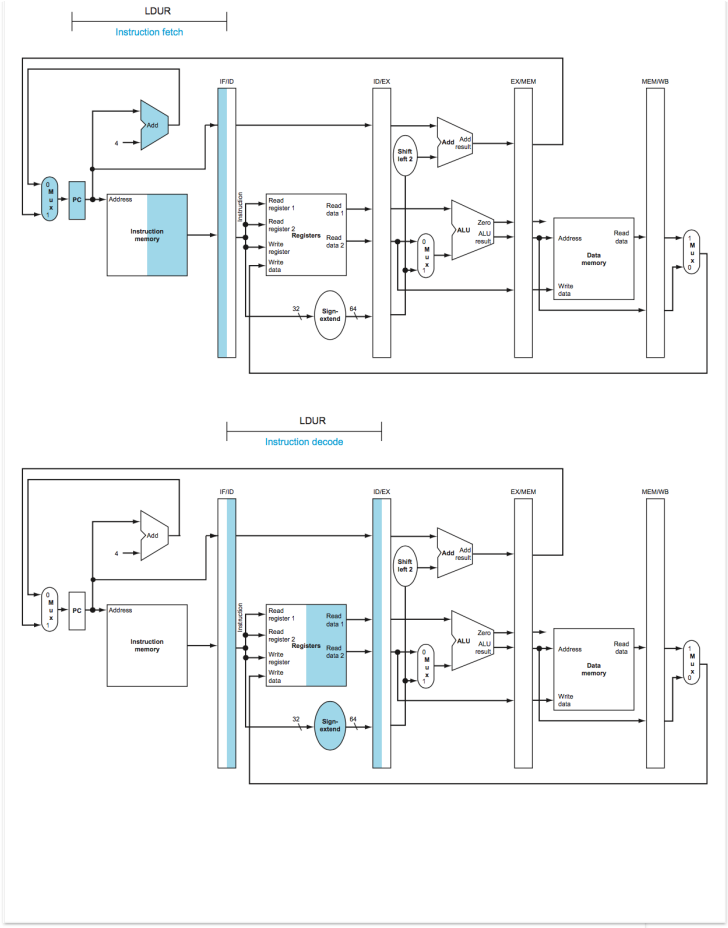
We show the instruction `LDUR` with the name of the pipe stage that is active in each figure. The five stages are the following:

1. *Instruction fetch:* The top portion of the figure below shows the instruction being read from memory using the address in the PC and then being placed in the IF/ID pipeline register. The PC address is incremented by 4 and then written back into the PC to be ready for the next clock cycle. This incremented address is also saved in the IF/ID pipeline register in case it is needed later for an instruction, such as `CBZ`. The computer cannot know which type of instruction is being fetched, so it must prepare for any instruction, passing potentially needed information down the pipeline.

Figure 4.6.3: IF and ID: First and second pipe stages of an instruction, with the active portions of the datapath in the figure above highlighted (COD Figure 4.35).

Shading on the right half of the register file or memory means the element is read in that stage, and shading of the left half means it is written in that stage.

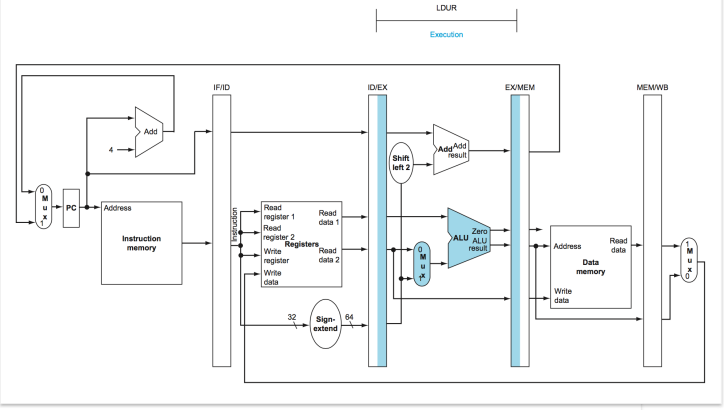
As in COD Section 4.2 (Logic design conventions), there is no confusion when reading and writing registers, because the contents change only on the clock edge. Although the load needs only the top register in stage 2, it doesn't hurt to do potentially extra work, so it sign-extends the constant and reads both registers into the ID/EX pipeline register. We don't need all three operands, but it simplifies control to keep all three.



2. *Instruction decode and register file read:* The bottom portion of the figure above shows the instruction portion of the IF/ID pipeline register supplying the immediate field, which is sign-extended to 64 bits, and the register numbers to read the two registers. All three values are stored in the ID/EX pipeline register, along with the incremented PC address. We again transfer everything that might be needed by any instruction during a later clock cycle.
3. *Execute or address calculation:* The figure below shows that the load instruction reads the contents of a register and the sign-extended immediate from the ID/EX pipeline register and adds them using the ALU. That sum is placed in the EX/MEM pipeline register.

Figure 4.6.4: EX: The third pipe stage of a load instruction, highlighting the portions of the datapath in COD Figure 4.34 (The pipelined version of the single-cycle datapath) used in this pipe stage (COD Figure 4.36).

The register is added to the sign-extended immediate, and the sum is placed in the EX/MEM pipeline register.

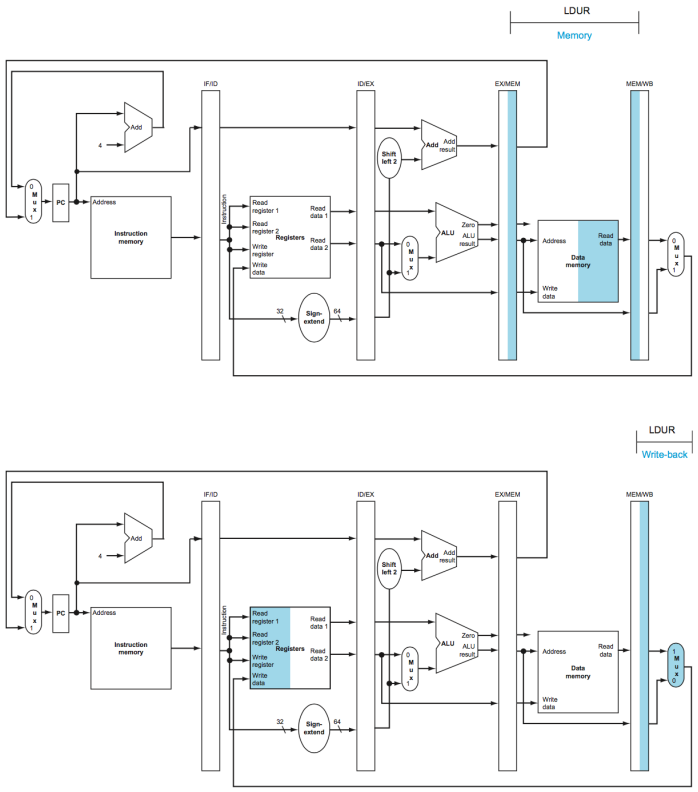


4. *Memory access:* The top portion of the figure below shows the load instruction reading the data memory using the address from the EX/MEM pipeline register and loading the data into the MEM/WB pipeline register.

Figure 4.6.5: MEM and WB: The fourth and fifth pipe stages of a load instruction, highlighting the portions of the datapath in COD Figure 4.34 (The pipelined version of the single-cycle datapath) used in this pipe stage (COD Figure 4.37).

Data memory is read using the address in the EX/MEM pipeline registers, and the data are placed in

the MEM/WB pipeline register. Next, data are read from the MEM/WB pipeline register and written into the register file in the middle of the datapath. Note: there is a bug in this design that is repaired in COD Figure 4.40 (The corrected pipelined datapath to handle the load instruction properly).



5. *Write-back:* The bottom portion of the figure above shows the final step: reading the data from the MEM/WB pipeline register and writing it into the register file in the middle of the figure.

This walk-through of the load instruction shows that any information needed in a later pipe stage must be passed to that stage via a pipeline register.

PARTICIPATION ACTIVITY 4.6.4: Pipelined load instruction walk-through.

Refer to the above load instruction (LDUR) pipe stages.

1) Which is NOT part of stage 1: IF (instruction fetch)?
   - ○ Instruction memory is read.
   - ○ The register file is written.
   - ○ The IF/ID register is written.

2) Which is NOT part of stage 2: ID (instruction decode)?
   - ○ PC is incremented.
   - ○ Register file is read.
   - ○ Sign extension.
   - ○ Control signals are generated for the current instruction.
   - ○ The IF/ID register is read, and the ID/EX register is written.

3) Which is NOT part of stage 3: EX (execution)?
   - ○ The ALU operates.
   - ○ Data memory is written.
   - ○ The ID/EX register is read, and the EX/MEM register is written.

4) Which is NOT part of stage 4: MEM (memory)?
   - ○ Instruction memory is read.
   - ○ Data memory is read.
   - ○ The EX/MEM register is read, and the MEM/WB register is written.

5) Which is NOT part of stage 5: WB (write back)?

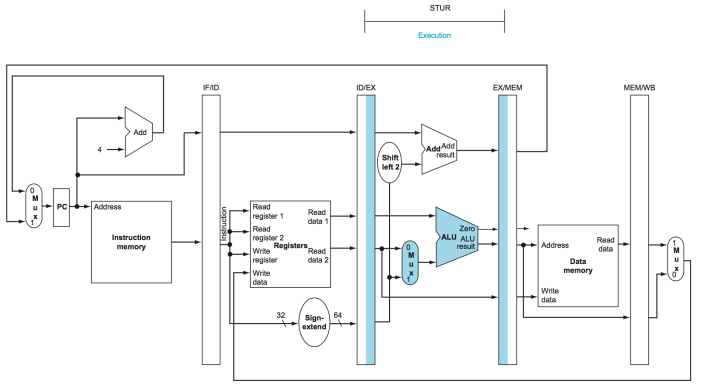
- ○ The register file is written.
- ○ The MEM/WB register is read, and the IF/ID register is written.

Walking through a store instruction shows the similarity of instruction execution to the load instruction, as well as passing the information for later stages. Here are the five pipe stages of the store instruction:

1. *Instruction fetch:* The instruction is read from memory using the address in the PC and then is placed in the IF/ID pipeline register. This stage occurs before the instruction is identified, so the top portion of COD Figure 4.35 (IF and ID: First and second pipe stages of an instruction ...) works for store as well as load.
2. *Instruction decode and register file read:* The instruction in the IF/ID pipeline register supplies the register numbers for reading two registers and extends the sign of the immediate operand. These three 64-bit values are all stored in the ID/EX pipeline register. The bottom portion of COD Figure 4.35 (IF and ID: First and second pipe stages of an instruction ...) for load instructions also shows the operations of the second stage for stores. These first two stages are executed by all instructions, since it is too early to know the type of the instruction. (While the store instruction uses the Rt field to read the second register in this pipe stage, that detail is not shown in this pipeline diagram, so we can use the same figure for both.)
3. *Execute and address calculation:* The figure below shows the third step; the effective address is placed in the EX/MEM pipeline register.

Figure 4.6.6: EX: The third pipe stage of a store instruction (COD Figure 4.38).

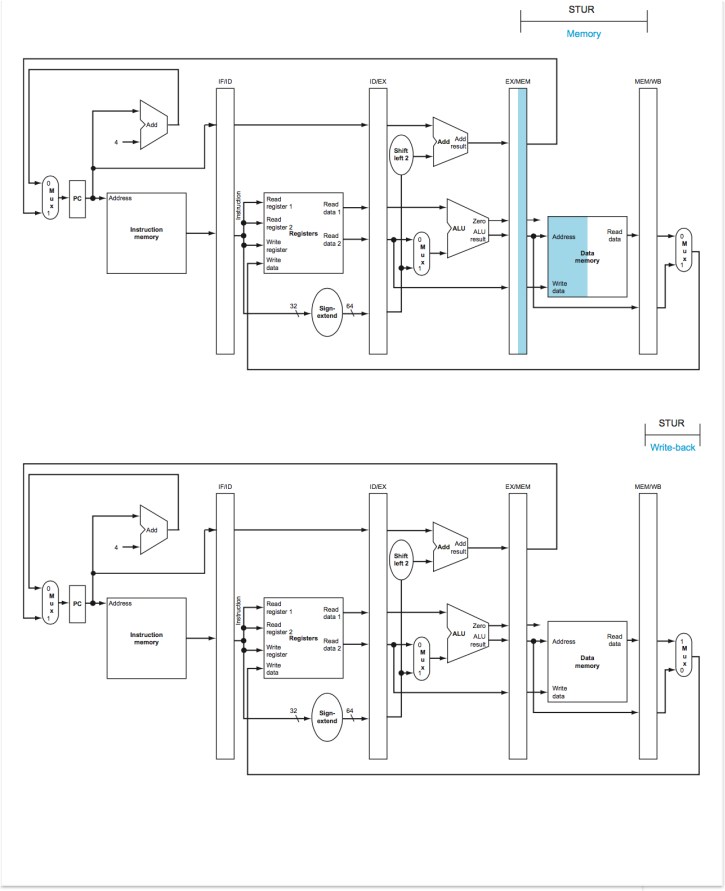
Unlike the third stage of the load instruction in COD Figure 4.36 (EX: The third pipe stage of a load instruction ...), the second register value is loaded into the EX/MEM pipeline register to be used in the next stage. Although it wouldn't hurt to always write this second register into the EX/MEM pipeline register, we write the second register only on a store instruction to make the pipeline easier to understand.



4. *Memory access:* The top portion of the figure below shows the data being written to memory. Note that the register containing the data to be stored was read in an earlier stage and stored in ID/EX. The only way to make the data available during the MEM stage is to place the data into the EX/MEM pipeline register in the EX stage, just as we stored the effective address into EX/MEM.

Figure 4.6.7: MEM and WB: The fourth and fifth pipe stages of a store instruction (COD Figure 4.39).

In the fourth stage, the data are written into data memory for the store. Note that the data come from the EX/MEM pipeline register and that nothing is changed in the MEM/WB pipeline register. Once the data are written in memory, there is nothing left for the store instruction to do, so nothing happens in stage 5.



5. *Write-back:* The bottom portion of the figure above shows the final step of the store. For this instruction, nothing happens in the write-back stage. Since every instruction behind the store is already in progress, we have no way to accelerate those instructions. Hence, an instruction passes through a stage even if there is nothing to do, because later instructions are already progressing at the maximum rate.

PARTICIPATION ACTIVITY 4.6.5: Store instruction pipeline stages.

1) The store and load instructions behave similarly in stage 1 (IF: instruction fetch).
   - ○ True
   - ○ False

2) The store and load instructions behave similarly in stage 2 (ID: instruction decode).
   - ○ True
   - ○ False

3) The store and load instructions behave similarly in stage 3 (EX: execute).
   - ○ True
   - ○ False

4) The store and load instructions both write the data memory in stage 4 (MEM: memory).
   - ○ True
   - ○ False

5) The store and load instructions differ in stage 5 (WB: write back) in that load writes to the register file whereas store reads the register file.
   - ○ True
   - ○ False

The store instruction again illustrates that to pass something from an early pipe stage to a later pipe stage, the information must be placed in a pipeline register; otherwise, the information is lost when the next instruction enters that pipeline stage. For the store instruction, we needed to pass one of the registers read in the ID stage to the MEM stage, where it is stored in memory. The data were first placed in the ID/EX pipeline register and then passed to the EX/MEM pipeline register.

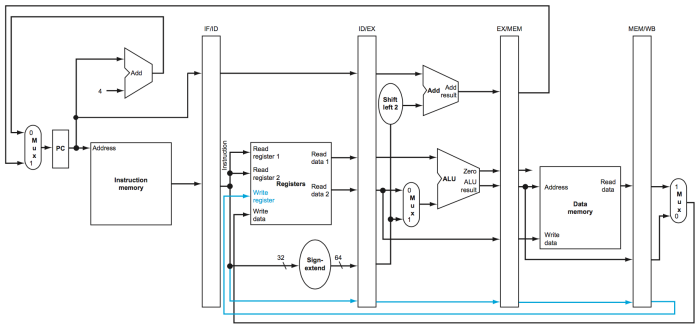
Load and store illustrate a second key point: each logical component of the datapath—such as instruction memory, register read ports, ALU, data memory, and register write port—can be used only within a *single* pipeline stage. Otherwise, we would have a *structural hazard*. Hence, these components, and their control, can be associated with a single pipeline stage.

Now we can uncover a bug in the design of the load instruction. Did you see it? Which register is changed in the final stage of the load? More specifically, which instruction supplies the write register number? The instruction in the IF/ID pipeline register supplies the write register number, yet this instruction occurs considerably *after* the load instruction!

Hence, we need to preserve the destination register number in the load instruction. Just as store passed the register *value* from the ID/EX to the EX/MEM pipeline registers for use in the MEM stage, load must pass the register *number* from the ID/EX through EX/MEM to the MEM/WB pipeline register for use in the WB stage. Another way to think about the passing of the register number is that to share the pipelined datapath, we need to preserve the instruction read during the IF stage, so each pipeline register contains a portion of the instruction needed for that stage and later stages.

Figure 4.6.8: The corrected pipelined datapath to handle the load instruction properly (COD Figure 4.40).

The write register number now comes from the MEM/WB pipeline register along with the data. The register number is passed from the ID pipe stage until it reaches the MEM/ WB pipeline register, adding five more bits to the last three pipeline registers. This new path is shown in color.



The figure above shows the correct version of the datapath, passing the write register number first to the ID/EX register, then to the EX/MEM register, and finally to the MEM/WB register. The register number is used during the WB stage to specify the register to be written.

PARTICIPATION ACTIVITY 4.6.6: Fixing the load instruction bug.

The datapath originally had a bug relating to the address of the register into which loaded data should be written, for a load instruction. Refer to the figure above, which illustrates the handling of that address for each cycle in the corrected datapath.

Written to ID/EX register | Written to IF/ID register | Written to EX/MEM register

Used as the register file's write address | Written to MEM/WB register

CC 1

CC 2

CC 3

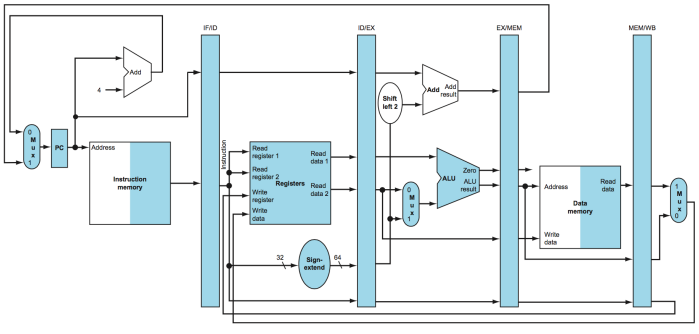
CC 4

CC 5

Reset

The figure below is a single drawing of the corrected datapath, highlighting the hardware used in all five stages of the load register instruction. See COD Section 4.8 (Control hazards) for an explanation of how to make the branch instruction work as expected.

Figure 4.6.9: The portion of the datapath in the figure above that is used in all five stages of a load instruction (COD Figure 4.41).



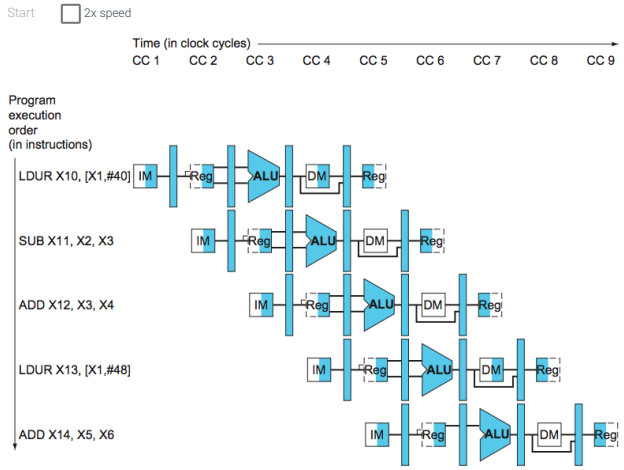
Graphically representing pipelines

Pipelining can be difficult to master, since many instructions are simultaneously executing in a single datapath in every clock cycle. To aid understanding, there are two basic styles of pipeline figures: *multiple-clock-cycle pipeline diagrams*, such as COD Figure 4.33 (Instructions being executed using the single-cycle datapath ...), and *single-clock-cycle pipeline diagrams*, such as COD Figures 4.35 (IF and ID: First and second pipe stages of an instruction ...). The multiple-clock-cycle diagrams are simpler but do not contain all the details. For example, consider the following five-instruction sequence:

```
LDUR   X10, [X1,#40]
SUB    X11, X2, X3
ADD    X12, X3, X4
LDUR   X13, [X1,#48]
ADD    X14, X5, X6
```

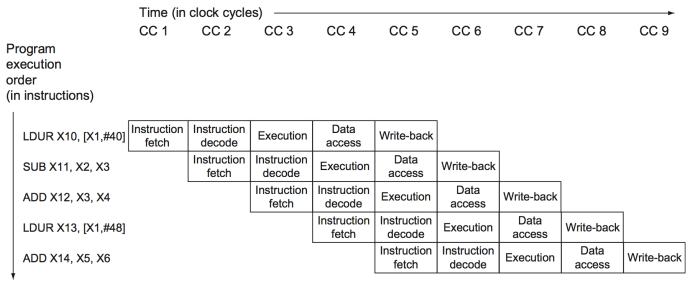
The animation below shows the multiple-clock-cycle pipeline diagram for these instructions. Time advances from left to right across the page in these diagrams, and instructions advance from the top to the bottom of the page, similar to the laundry pipeline in COD Figure 4.24 (The laundry analogy for pipelining). A representation of the pipeline stages is placed in each portion along the instruction axis, occupying the proper clock cycles. These stylized datapaths represent the five stages of our pipeline graphically, but a rectangle naming each pipe stage works just as well. COD Figure 4.43 (Traditional multiple-clock-cycle pipeline diagram ...) shows the more traditional version of the multiple-clock-cycle pipeline diagram. Note that the animation below shows the physical resources used at each stage, while COD Figure 4.43 (Traditional multiple-clock-cycle pipeline diagram ...) uses the *name* of each stage.

PARTICIPATION ACTIVITY 4.6.7: Multiple-clock-cycle pipeline diagram of five instructions (COD Figure 4.42).



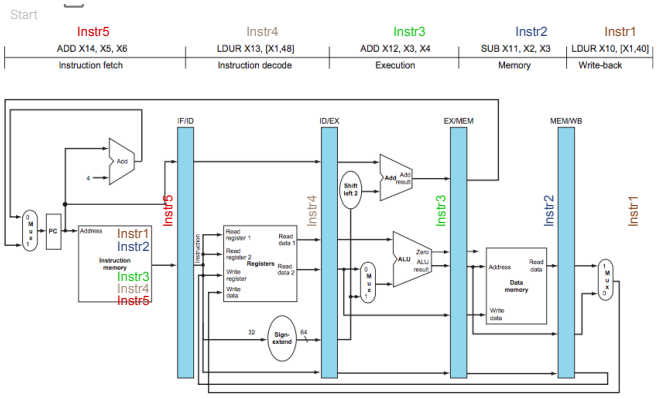
The above style of pipeline representation shows the complete execution of instructions in a single figure. Instructions are listed in instruction execution order from top to bottom, and clock cycles move from left to right. Unlike COD Figure 4.27 (Graphical representation of the instruction pipeline), here we show the pipeline registers between each stage. The figure below shows the traditional way to draw this diagram.

Figure 4.6.10: Traditional multiple-clock-cycle pipeline diagram of five instructions in the figure above (COD Figure 4.43).



Single-clock-cycle pipeline diagrams show the state of the entire datapath during a single clock cycle, and usually all five instructions in the pipeline are identified by labels above their respective pipeline stages. We use this type of figure to show the details of what is happening within the pipeline during each clock cycle; typically, the drawings appear in groups to show pipeline operation over a sequence of clock cycles. We use multiple-clock-cycle diagrams to give overviews of pipelining situations. (COD Section 4.13 (Advanced topic: An introduction to digital design ...) gives more illustrations of single-clock diagrams if you would like to see more details about COD Figure 4.42 (Multiple-clock-cycle pipeline diagram ...).) A single-clock-cycle diagram represents a vertical slice of one clock cycle through a set of multiple-clock-cycle diagrams, showing the usage of the datapath by each of the instructions in the pipeline at the designated clock cycle. For example, the animation below shows the single-clock-cycle diagram corresponding to clock cycle 5 of COD Figures 4.42 (Multiple-clock-cycle pipeline diagram ...) and 4.43 (Traditional multiple-clock-cycle pipeline diagram ...). Obviously, the single-clock-cycle diagrams have more detail and take significantly more space to show the same number of clock cycles. The exercises ask you to create such diagrams for other code sequences.

PARTICIPATION ACTIVITY 4.6.8: The single-clock-cycle diagram corresponding to clock cycle 5 of the pipeline in previous two figures (COD Figure 4.44).



PARTICIPATION ACTIVITY 4.6.9: Single-clock-cycle diagram.

Consider the above figure showing clock cycle 5 of the pipeline for a sequence of five instructions. Indicate the status of the instruction provided.

1) Instr5 (ADD X14...)
   - ○ Being fetched
   - ○ Being decoded
   - ○ Executing on the ALU

2) Instr3 (ADD X12...)
   - ○ Being fetched
   - ○ Being decoded
   - ○ Executing on the ALU

3) Instr1 (LDUR X10...)
   - ○ Being fetched
   - ○ Writing back
   - ○ Nothing; already done

4) Consider CC 6 (clock cycle 6). What is Instr1 (LDUR X10...) doing?
   - ○ Writing back
   - ○ Nothing; already done

5) Consider CC 6 (clock cycle 6). Assume Instr1 to Instr10 exist with no branches. What instruction is being fetched?
   - ○ Instr1
   - ○ Instr5
   - ○ Instr6

6) In what CC will Instr5 NOT be in the pipeline?
   - ○ CC 5
   - ○ CC 8
   - ○ CC 10

PARTICIPATION ACTIVITY 4.6.10: Check yourself: Pipelining.

A group of students were debating the efficiency of the five-stage pipeline when one student pointed out that not all instructions are active in every stage of the pipeline. After deciding to ignore the effects of hazards, they made the following statements. Which are correct?

1) Allowing branches and ALU instructions to take fewer stages than the five required by the load instruction will increase pipeline performance under all circumstances.
   - ○ Correct
   - ○ Incorrect

2) You cannot make ALU instructions take fewer cycles because of the write-back of the result, but branches can take fewer cycles, so there is some opportunity for improvement.
   - ○ Correct
   - ○ Incorrect

> In the 6600 Computer, perhaps even more than in any previous computer, the control system is the difference.
> *James Thornton, Design of a Computer: The Control Data 6600, 1970.*

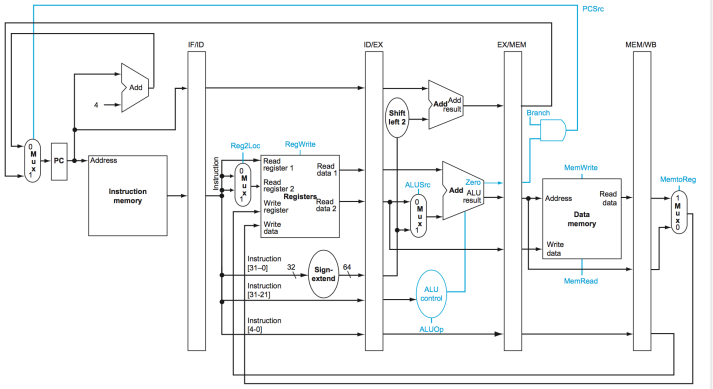
Pipelined control

Just as we added control to the single-cycle datapath in COD Section 4.3 (Building a datapath), we now add control to the pipelined datapath. We start with a simple design that views the problem through rose-colored glasses.

The first step is to label the control lines on the existing datapath. The figure below shows those lines. We borrow as much as we can from the control for the simple datapath. In particular, we use the same ALU control logic, branch logic, read-register2-number multiplexor, and control lines. These functions are defined in COD Figures 4.12 (How the ALU control bits are set …), 4.16 (The effect of each of the seven control signals), and 4.18 (The setting of the control lines …). We reproduce the key information in the tables below to make the following discussion easier to absorb.

Figure 4.6.11: The corrected pipelined datapath with the control signals identified. (COD Figure 4.45).

This datapath borrows the control logic for PC source, register destination number, and ALU control from COD Section 4.4 (A simple implementation scheme). Note that we now need the opcode field of the instruction in the EX stage as input to ALU control, so these bits must also be included in the ID/EX pipeline register.



As was the case for the single-cycle implementation, we assume that the PC is written on each clock cycle, so there is no separate write signal for the PC. By the same argument, there are no separate write signals for the pipeline registers (IF/ ID, ID/EX, EX/MEM, and MEM/WB), since the pipeline registers are also written during each clock cycle.

To specify control for the pipeline, we need only set the control values during each pipeline stage. Because each control line is associated with a component active in only a single pipeline stage, we can divide the control lines into five groups according to the pipeline stage.

1. *Instruction fetch:* The control signals to read instruction memory and to write the PC are always asserted, so there is nothing special to control in this pipeline stage.
2. *Instruction decode/register file read:* We need to select the correct register number for read register 2, so the signal Reg2Loc is set. The signal selects instruction bits 20:16 (Rm) or 4:0 (Rt).
3. *Execution/address calculation:* The signals to be set are ALUOp and ALUSrc (see figure below). The signals select the ALU operation and either Read data 2 or a sign-extended immediate as inputs to the ALU.

Figure 4.6.12: How the ALU control bits are set depends on the ALUOp control bits and the different opcodes for the R-type instruction (COD Figure 4.46).

Note: Copy of COD Figure 4.12

This figure shows how the ALU control bits are set depending on the ALUOp control bits and the different opcodes for the R-type instruction.

Instruction	ALUOp	Instruction operation	Opcode	Desired ALU action	ALU control input
LDUR	00	load register	XXXXXXXXXXX	add	0010
STUR	00	store register	XXXXXXXXXXX	add	0010
CBZ	01	compare and branch on zero	XXXXXXXXXXX	pass input b	0111
R-type	10	ADD	10001011000	add	0010
R-type	10	SUB	11001011000	subtract	0110
R-type	10	AND	10001010000	AND	0000
R-type	10	ORR	10101010000	OR	0001

Figure 4.6.13: Copy of the table showing the effect of each of the seven control signals (COD Figure 4.47).

Note: Copy of COD Figure 4.16

The function of each of seven control signals is defined. The ALU control lines (ALUOp) are defined in the second column of the figure above. When a 1-bit control to a two-way multiplexor is asserted, the multiplexor selects the input corresponding to 1. Otherwise, if the control is deasserted, the multiplexor selects the 0 input. Note that PCSrc is controlled by an AND gate in COD Figure 4.45 (The pipelined datapath of Figure 4.40 with the control signals identified). If the Branch signal and

the ALU Zero signal are both set, then PCSrc is 1; otherwise, it is 0. Control sets the Branch signal only during a `CBZ` instruction; otherwise, PCSrc is set to 0.

Signal name	Effect when deasserted (0)	Effect when asserted (1)
Reg2Loc	The register number for Read register 2 comes from the Rm field (bits 20:16).	The register number for Read register 2 comes from the Rt field (bits 4:0).
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the immediate field of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory

4. *Memory access:* The control lines set in this stage are Branch, MemRead, and MemWrite. The compare and branch on zero, load, and store instructions set these signals, respectively. Recall that PCSrc in the figure above selects the next sequential address unless control asserts Branch and the ALU result was 0.
5. *Write-back:* The two control lines are MemtoReg, which decides between sending the ALU result or the memory value to the register file, and Reg-Write, which writes the chosen value.

Since pipelining the datapath leaves the meaning of the control lines unchanged, we can use the same control values. The figure below has the same values as in COD Section 4.4 (A simple implementation scheme), but now the nine control lines are grouped by pipeline stage.

Figure 4.6.14: Copy of the table showing the setting of the control lines is completely determined by the opcode fields of the instruction (COD Figure 4.48).

The values of the control lines are the same as in COD Figure 4.18 (The setting of the control lines is completely determined by the opcode fields of the instruction), but they have been shuffled into four groups corresponding to the last four pipeline stages.

	Instruction decode stage control lines	Execution/address calculation stage control lines			Memory access stage control lines			Write-back stage control lines	
Instruction	Reg2Loc	ALUOp1	ALUOp0	ALUSrc	Branch	MemRead	MemWrite	RegWrite	MemtoReg
R-format	0	1	0	0	0	0	0	1	0
LDUR	X	0	0	1	0	1	0	1	1
STUR	1	0	0	1	0	0	1	0	X
CBZ	1	0	1	0	1	0	0	0	X

Implementing control means setting the nine control lines to these values in each stage for each instruction. We need to set Reg2Loc to read the right register while we are still decoding the instruction during the ID pipeline stage. We need to use bits of the opcode to control this multiplexor, just as we did for sign extension as explained in the elaboration earlier. If you look at the opcodes carefully—such as in Figure 4.22 (The control function for the simple single-cycle implementation ...)—you can see that instruction bit 28 is a 0 for R-format instructions and a 1 for the rest. This is exactly what we need to control the register address multiplexor, so we simply connect instruction bit 28 to Reg2Loc.

Since the rest of the control lines start with the EX stage, we can create the control information during instruction decode for the later stages. The simplest way to pass these control signals is to extend the pipeline registers to include control information. The figure below shows that these control signals are then used in the appropriate pipeline stage as the instruction moves down the pipeline, just as the destination register number for loads moves down the pipeline in COD Figure 4.40 (The corrected pipelined datapath ...). COD Figure 4.50 (The pipelined datapath of Figure 4.45, with the control signals ...) shows the full datapath with the extended pipeline registers and with the control lines connected to the proper stage along with the instruction bit 28 controlling the Reg2Loc register address multiplexor. (COD Section 4.13 (Advanced topic: An introduction to digital design ...) gives more examples of LEGv8 code executing on pipelined hardware using single-clock diagrams, if you would like to see more details.)

Figure 4.6.15: The eight control lines for the final three stages (COD Figure 4.49).

Note that three of the eight control lines are used in the EX phase, with the remaining five control lines passed on to the EX/MEM pipeline register extended to hold the control lines; three are used during the MEM stage, and the last two are passed to MEM/ WB for use in the WB stage.

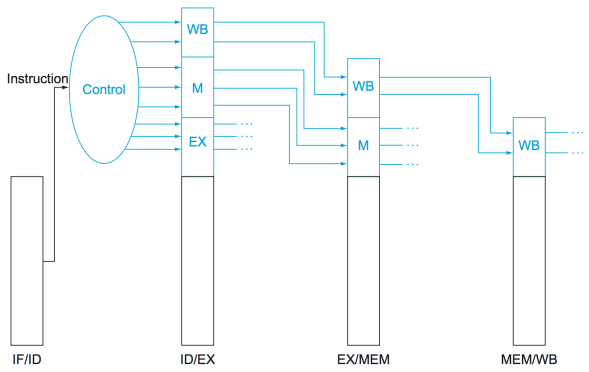
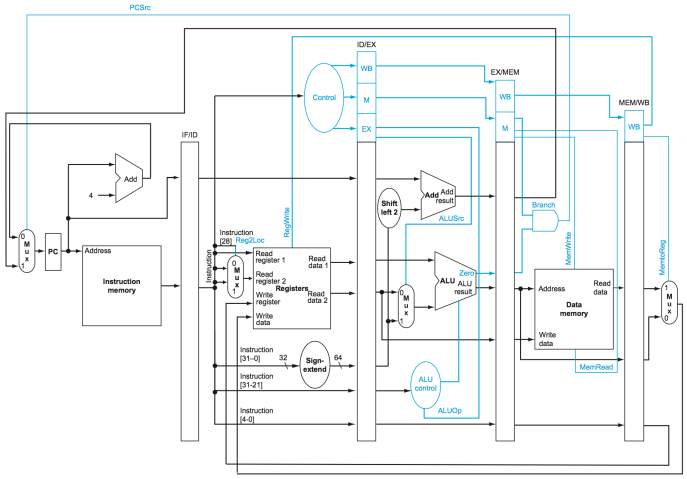


Figure 4.6.16: The corrected pipelined datapath with the control signals connected of the control portions of the pipelined registers (COD Figure

connected to the control portions of the pipelined registers (COD Figure 4.50).

The control values for the last three stages are created during the instruction decode stage and then placed in the ID/EX pipeline register. The control lines for each pipe stage are used, and remaining control lines are then passed to the next pipeline stage.



PARTICIPATION ACTIVITY 4.6.11: Pipelined control.

1) In the non-pipelined datapath, for an R-type instruction, all the needed values for control lines like ALUOp, ALUSrc, and RegWrite, can be determined when the instruction appears from instruction memory.
   - True
   - False

2) In the non-pipelined datapath, for a R-type instruction, the control signals ALUOp and ALUSrc are used in the instruction's 3rd clock cycle, while RegWrite is used in the instruction's 5th clock cycle.
   - True
   - False

3) In the pipelined datapath, for an R-type instruction, all the needed values for control lines like ALUOp, ALUSrc, and RegWrite, can be determined when the instruction appears from instruction memory.
   - True
   - False

4) In the pipelined datapath, for a R-type instruction, the control signals ALUOp and ALUSrc are used in the instruction's 3rd clock cycle, while RegWrite is used in the instruction's 5th clock cycle.
   - True
   - False

5) In the pipelined datapath, because the control signals are generated in an earlier cycle than when used, the signals must be stored in registers.
   - True
   - False

6) The LEGv8 pipelined control approach determines all control line values during an instruction's 1st clock cycle, the instruction fetch stage.
   - True
   - False

7) The control line values are different in the pipelined datapath than in the non-pipelined datapath.

True

False

Elaboration

Because one of the source registers for LEGv8 instructions is found in different places in different instructions, the LEGv8 computer designer has a more difficult challenge that the MIPS computer designer, where the source registers are always in the same location in the MIPS instruction formats. Thus, the MIPS decode stage can read two registers without doing any decoding. In the ID stage for LEGv8, either you need a three-ported register file to read the three possible registers, or you need to do some partial decoding, as we have done in this chapter.

Provide feedback on this section