Pipelined Datapath and control

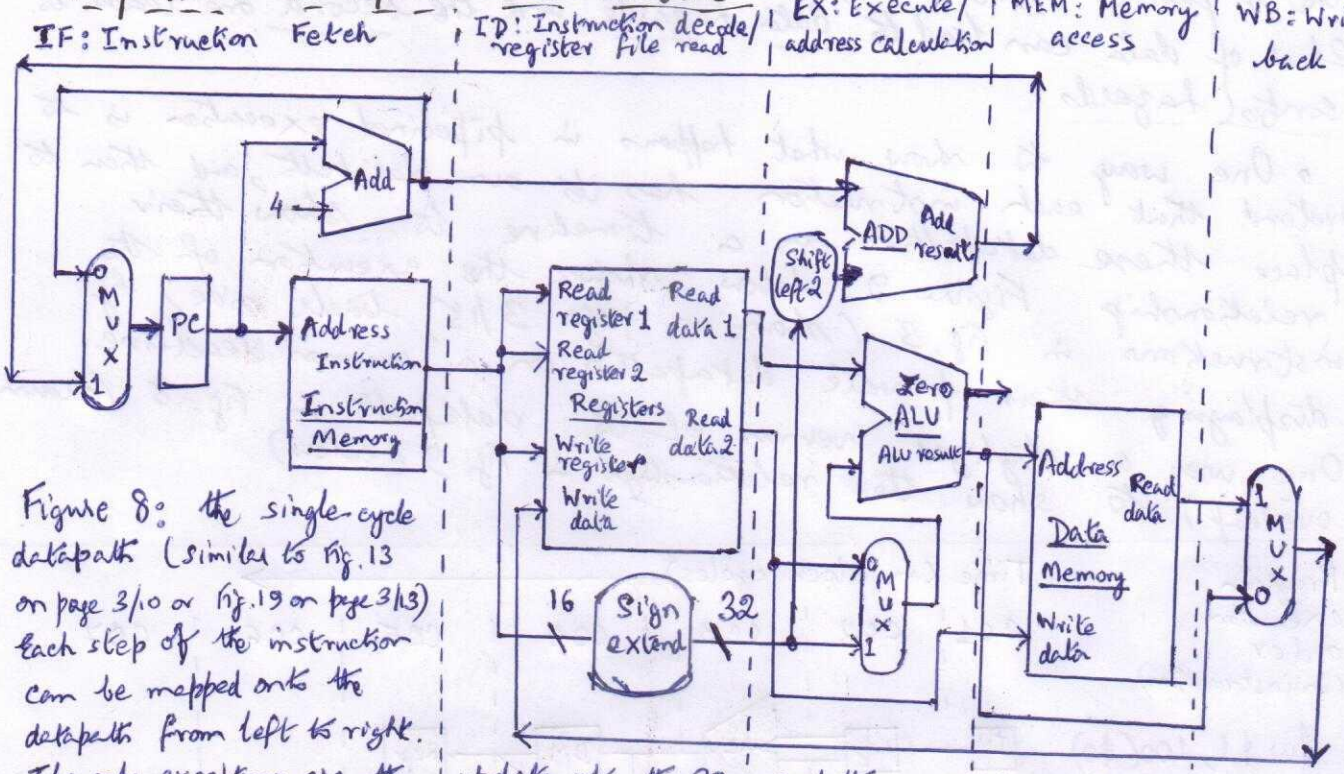


Figure 8: the single-cycle datapath (similar to Fig. 13 on page 3/10 or Fig. 19 on page 3/13) Each step of the instruction can be mapped onto the datapath from left to right. The only exceptions are the update of the PC and the write back step, which sends either the ALU result or the data from memory to the left to be written into the register file.

- Figure 8 shows the single-cycle datapath (studied earlier from page 3/16 to page 3/13) 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, one must divide/separate the datapath into five segments/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; and 5) WB: Write back.
- In Fig. 8, these five components correspond to the way the datapath is drawn; instructions and data move generally from left to right through the five stages as they complete execution. There are, however, two exceptions to this left-to-right flow of instructions:
  a) the write-back stage, which places the result back into the register file in the middle of the datapath
  b) the selection of the next value of the PC, choosing between the incremented PC and the branch address from the MEM stage.

Note that data flowing from right to left does not affect the current instruction;

P.T.O.

these reverse data movements influence only the later instructions in the pipeline. It may be observed that the first right-to-left flow of data can lead to data hazards and the second one 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. Figure 9 below shows the execution of the instructions in Fig. 3 (shown on page 3/15, back side) by displaying their private datapaths on a common deadline. One uses a stylized version of the datapath in Fig. 8 (shown overleaf) to show the relationships in Fig 9 (below)

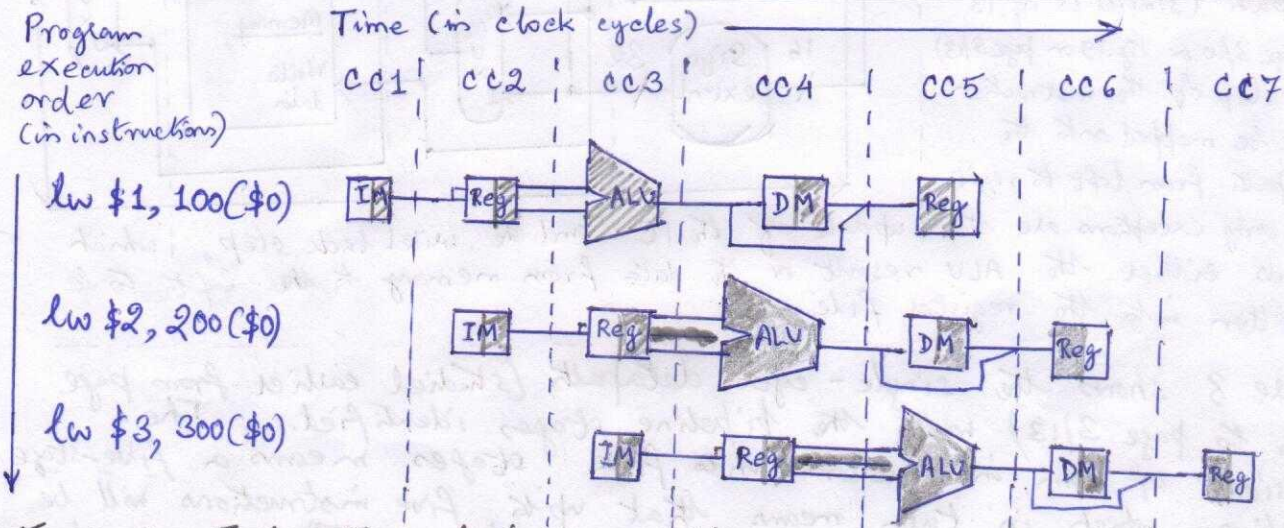


Figure 9: Instructions being executed using the single-cycle datapath of Fig. 8, assuming pipelined execution. Similar to the earlier figures, this figure pretends that each instruction has its own datapath, and shades each portion according to use. Unlike those figures, each stage is labeled with the physical resource used in that stage (corresponding to the portions of the datapath in Fig. 8). '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 shown by drawing the unshaded left half of the register file in ID stage, when it is not being written, and the unshaded right half in WB stage, when it is not being read. As before, it is assumed that register file is written in first half of clock cycle, and register file is read during second half.

Pipelined Datapaths and control (cont.)

Figure 9 (on backside of page 3/21) apparently suggests that three instructions need three datapaths. Instead, one adds registers to hold data so that portions of a single datapath can be shared during instruction execution.

For example, as Fig. 9 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 other pipeline stages, so one must place registers wherever there are dividing lines between stages in Fig. 8.

- Figure 10 (below) shows the pipelined datapaths 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.

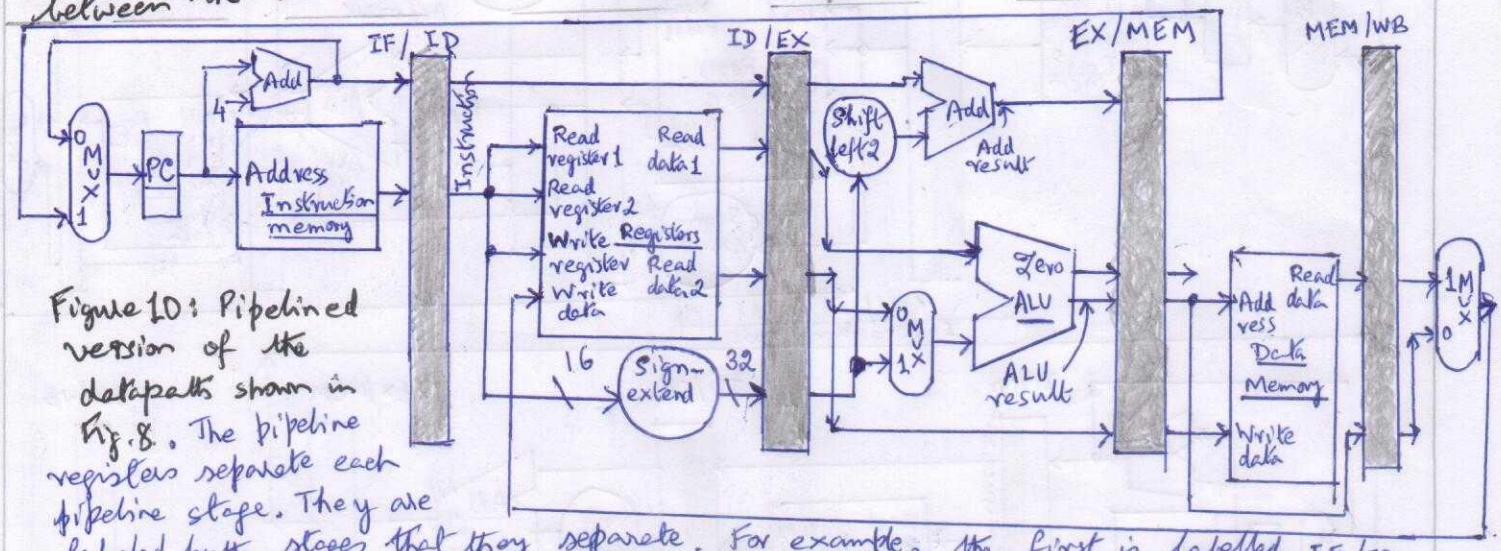


Figure 10: Pipelined version of the datapaths shown in Fig. 8. The pipeline registers separate each pipeline stage. They are labeled by the stages that they separate. For example, the first is labelled IF/ID because it separates the instruction fetch and instruction decode stages. The registers must be wide enough to store all the data corresponding to the signal lines that go through them. For example, the IF/ID register must be 64 bits wide, as it must hold both the 32-bit instruction fetched from memory and the incremented 32-bit PC address. The pipeline registers ID/EX, ~~EX/MEM~~ EX/MEM and MEM/WB contain 128, 97, and 64 bits, respectively.

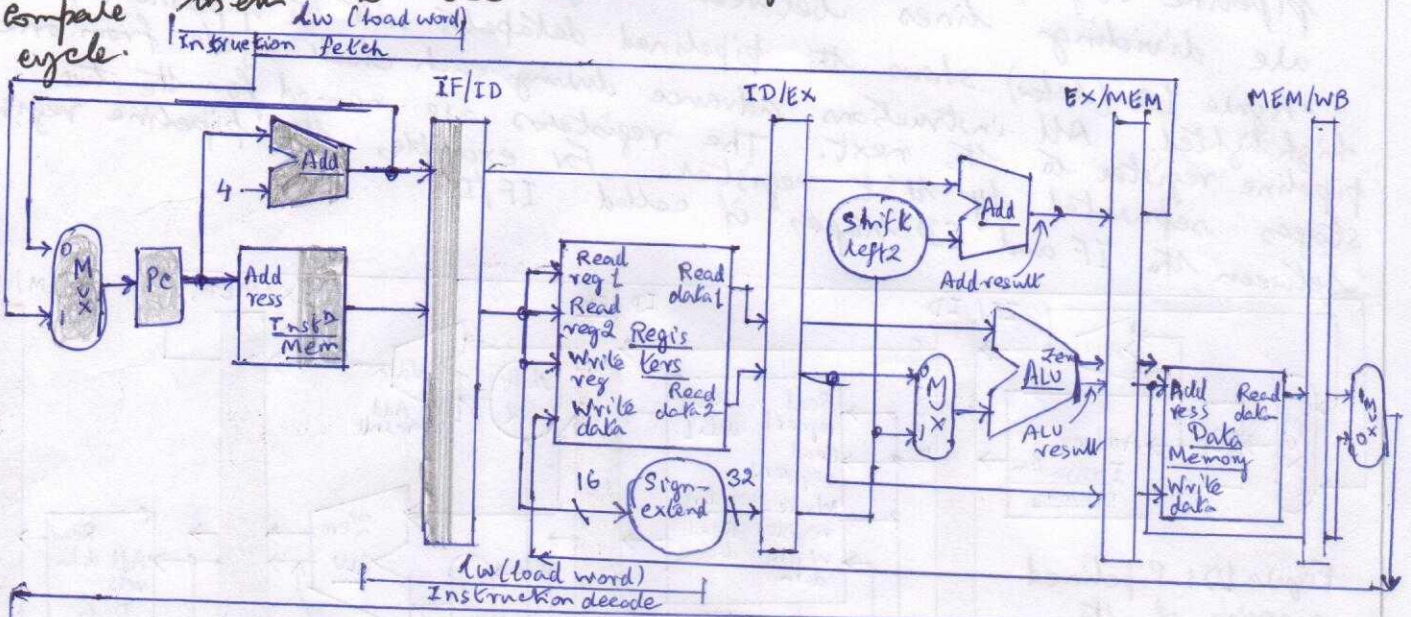
- Notice that there is no pipeline register at the end of the write-back stage. All instructions must update some state in the processor – the register file, memory, or the PC – so a separate pipeline register is redundant to the state that is updated. For example, a 'load'

P.T.O.

instruction will place its result in 1 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 (program counter) can be considered to be a pipeline register; one that feeds the IF stage of the pipeline. Unlike the shaded pipeline registers in Fig. 10, 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

To show how pipelining works, one can show sequences of figures to demonstrate operation over time. One can compare them to see what changes occur in each clock cycle.



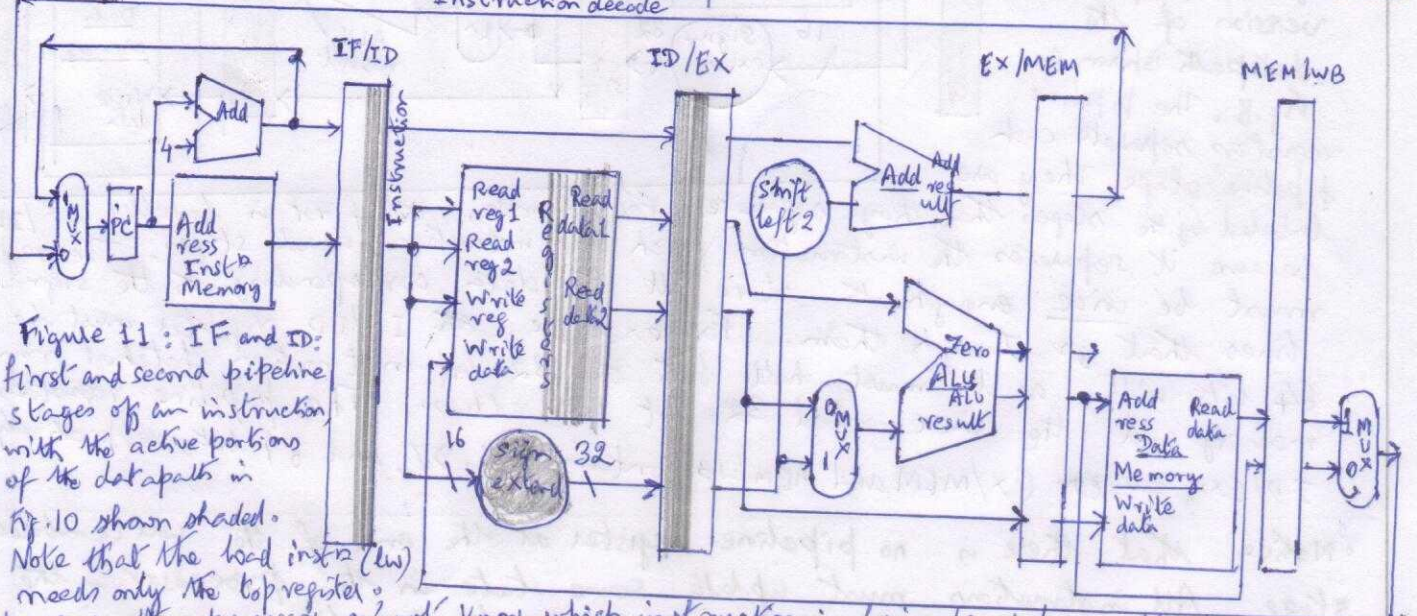


Figure 11: IF and ID: first and second pipeline stages of an instruction, with the active portions of the datapath in Fig. 10 shown shaded. Note that the load instr^n (lw) needs only the top register. However, the processor cannot know which instruction is being decoded; so, it sign-extends the 16-bit constant and reads both registers into the ID/EX pipeline register. Though one does not need all three operands, it simplifies the control to keep all three.

• In figure 11, the active portions of the datapath are shown shaded as a load instruction goes through the five stages of pipelined execution. Recall a load instruction ('lw') is active in all five stages. The right half of registers or memory are shown as shaded when they are being read and the left half is shown shaded when they are being written. The five pipeline stages are as follows:

1) Instruction fetch: the top portion of figure 11 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, eg. beq (beq $s1, $s2, 20 ⇒ if ($s1 == $s2) then go to pc+4+4*20). As the processor cannot know which type of instruction is being fetched, so it must prepare for any instruction, passing potentially needed information down the pipeline.

2) Instruction decode and register file read: the bottom portion of fig. 11 shows the instruction portion of the IF/ID pipeline register supplying the 16-bit immediate field, which is sign-extended to 32 bits, and the register numbers to read the two registers. All three values (consider lw $s1, 20($s2) ⇒ $s1 = Mem[$s2 + 20]) are stored in the ID/EX pipeline register, along with the incremented PC address. One again transfers everything that may be needed by any instruction during a later clock cycle.

3) Execute or address calculation: figure 12 (overleaf) shows that the load instruction reads the contents of register 1 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.

4) Memory access: The left-side portion of figure 13 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.

5) Write-back: The right-side portion of figure 13 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 datapath).

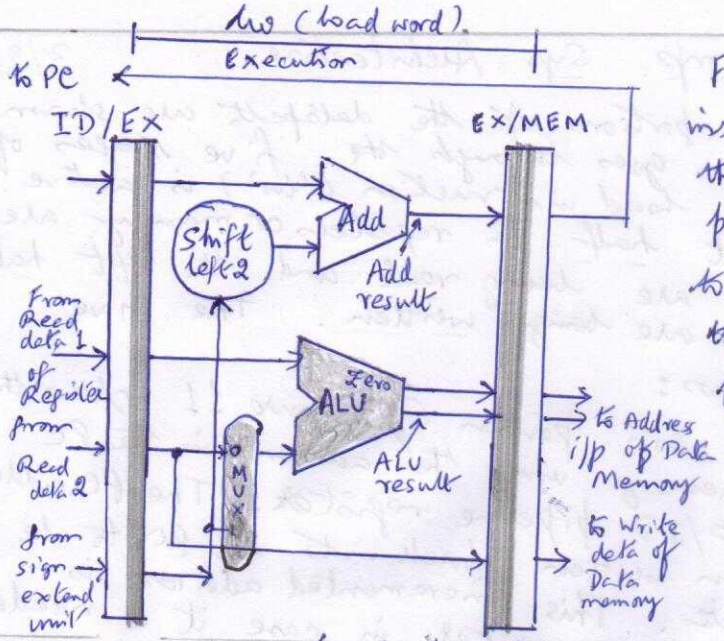
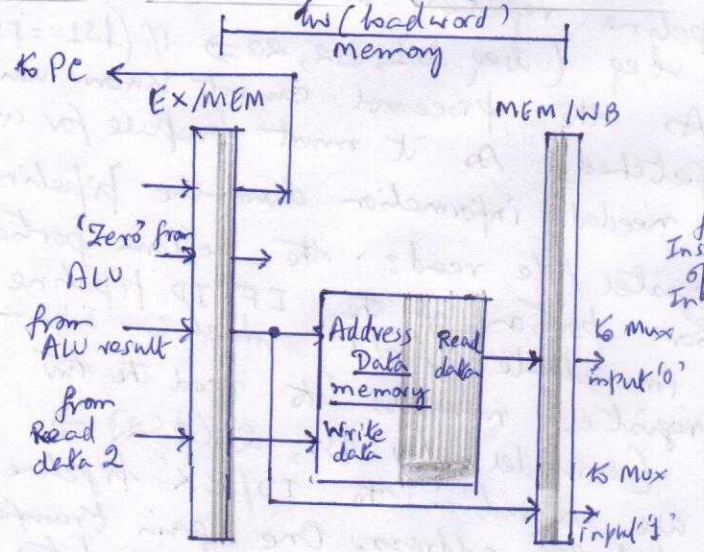


Figure 12: EX: the third pipe stage of a 'load' instruction, highlighting the portions of the datapaths in Fig. 10 used in this pipe stage. The register is added to the sign-extended immediate, and the sum is placed in the EX/MEM pipeline register.



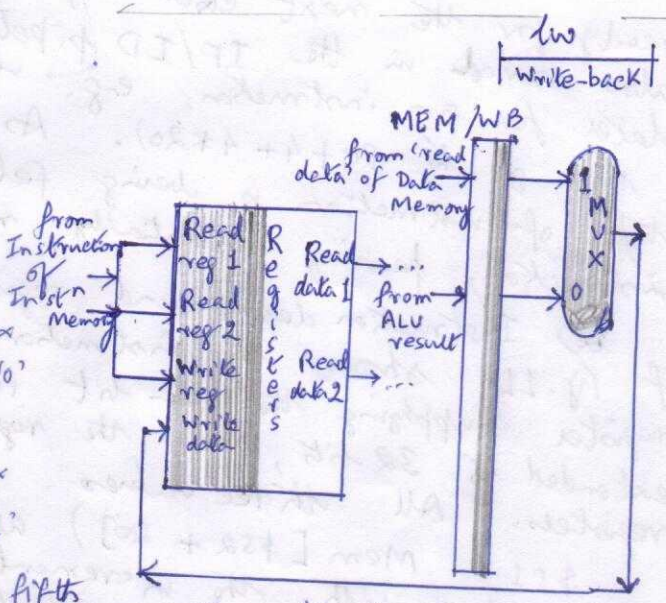


Figure 13: MEM and WB: the fourth and fifth pipe stages of a 'load' instruction, highlighting the portions of the datapaths in Figure 10 used in these pipe stages. Data memory is read using the address in the EX/MEM pipeline register, and the data is placed in the MEM/WB pipeline register. Next, data is read from the MEM/WB pipeline register and written into the register file in the middle of the datapath.

• One can similarly show the active portions of the datapaths highlighted as a store instruction goes through the five stages of pipelined execution. Small differences exist as follows:

1) unlike the third stage of the 'load' instruction, for the execution of 'store' instruction, the second register value to be loaded into the EX/MEM pipeline register to be used in the next stage. (note: store word SW $S1, 20($S2) ⇒ Memory[$S2 + 20] = $S1) needs For 'load' instruction, only the effective address (calculated by ALU) is placed in the EX/MEM pipeline or fifth register.

2) For 'store' instruction, nothing happens in the write-back stage. (as the contents ~~Store~~ of $S1 was already written in data memory in the 4th (MEM) stage. Since every instruction behind the 'store' instr is already in progress, one has no way to accelerate these instructions. Hence, ~~as~~ an instruction passes through a stage even if there is nothing to do, because the later instructions are already progressing at the maximum rate

• Execution of both 'load' and 'store' indicates that to pass something from an early pipe stage to a later pipe stage, the information must be placed in a pipeline register; else, the information is lost when the next instruction enters that pipeline stage. For the store instruction, one needed to pass one of the registers read in the ID stage to the MEM stage, where it is stored in memory. The data was first placed in the ID/EX pipeline register, and then passed to the EX/MEM pipeline register.

• A second key point illustrated by load and store: each logical component of the datapaths (such as instruction memory, register read ports, ALU, data memory, and register write port) can be used only within a single pipeline stage. Otherwise, one would have a structural hazard

• To identify a bug in the design of the 'load' instruction.
ld $s1, 20($2) ⇒ $s1 = M[$2+20) Which instruction supplies the write register number? The instruction in the IF/ID pipeline register supplies the write register number; however, this instruction occurs much after the 'load' instruction

Hence, one needs to preserve the destination register number in the 'load' instruction. The 'load' must pass the write register number from the IF/ID through ID/EX through EX/MEM to the MEM/WB pipeline registers for use in the WB stage. Thus, one needs to preserve the instruction read during the IF stage, so that each pipeline register contains a portion of the instruction needed for that stage and later stages.

Graphically Representing Pipelines

- Pipelining can be difficult to understand, since many instructions are simultaneously executing in a single datapath in every clock cycle. There exist two basic styles of pipeline diagrams (as an aid to understanding), which are multiple-clock-cycle pipeline diagrams (such as Fig. 9 on back side of page 3/21), and single-clock-cycle pipeline diagrams (such as figures 10 to 13). The multiple-clock-cycle diagrams are simpler but do not contain all the details. For example, consider the following five-instruction sequence:

```
lw  $10, 20($1)
sub $11, $2, $3
add $12, $3, $4
lw  $13, 24($1)
add $14, $5, $6
```

Figure 14 (below) shows the multiple-clock-cycle pipeline diagram for these instructions. Time advances from left to right across the page in these diagrams, while instructions advance from the top to the bottom of the page. A representation of pipeline stages is placed in each portion along the instruction axis, occupying the proper clock cycles.

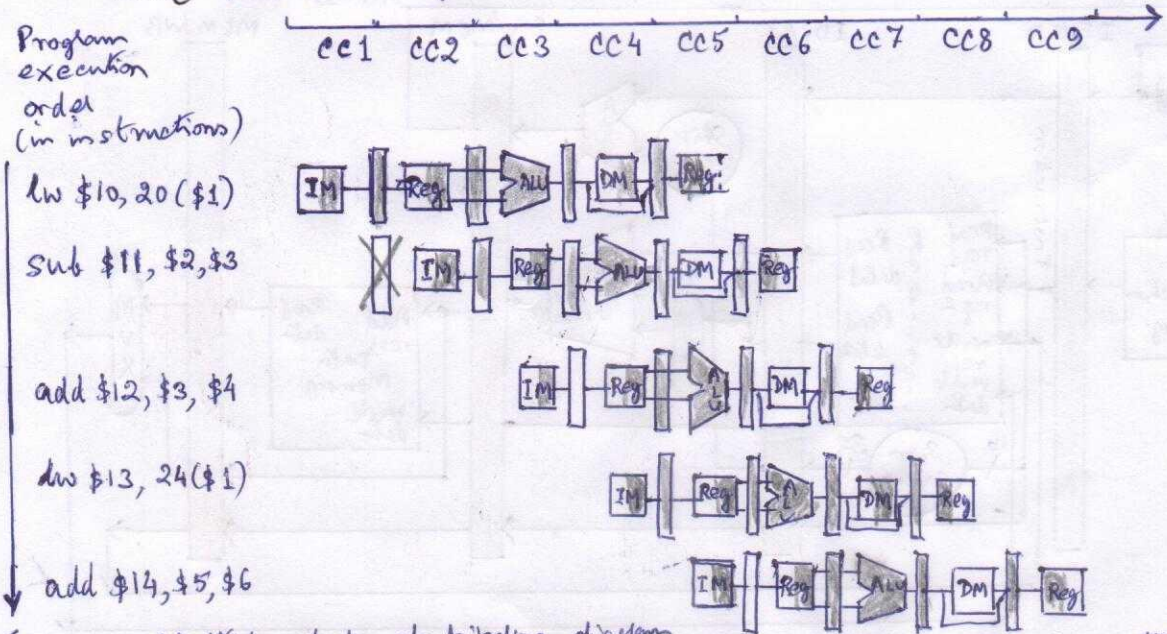


Figure 14: Multiple-clock-cycle pipeline diagram of five instructions: this style of pipeline representation shows the complete execution of instructions in a single figure. The pipeline registers are shown between each pair of stages.

In fig 14 above, the stylized datapaths represent the five stages of our pipeline graphically. The physical resources used at each stage are shown in fig. 14; whereas, in fig 15 (overleaf), which shows the more traditional version of multiple-cycle pipeline diagram, the name of each stage is used.

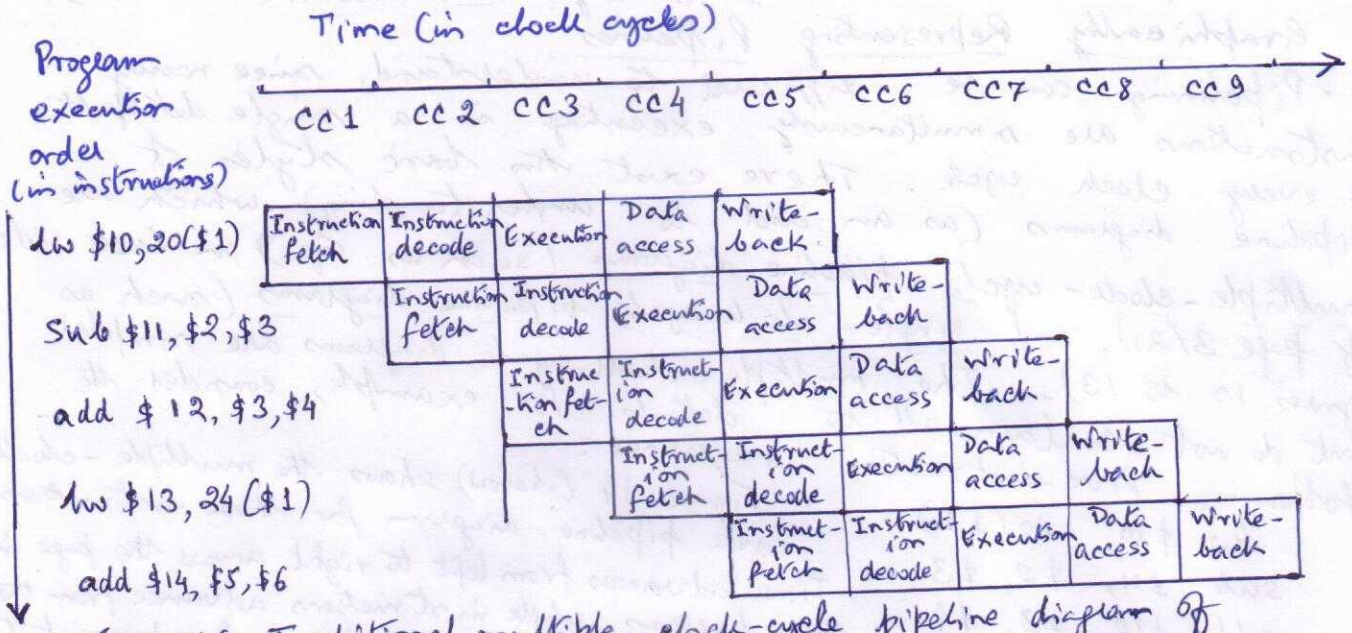


Figure 15: Traditional multiple-clock-cycle pipeline diagram of five instructions portrayed in Fig. 14.

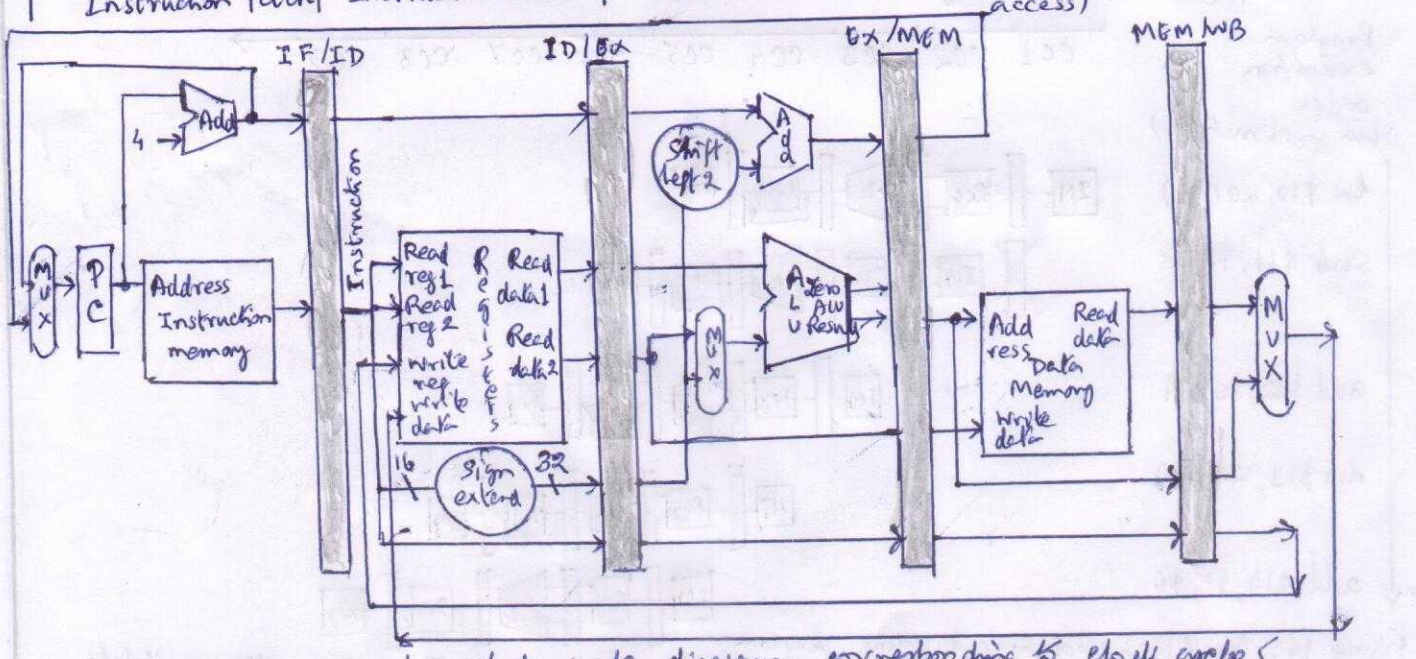


Figure 16: The single-clock-cycle diagram corresponding to clock cycle 5 (CC5) of the pipeline in Figures 14 and 15. As one can see, a single-clock-cycle figure is a vertical slice through a multiple-clock-cycle diagram.

Pipelined Control

• To add control to the pipelined datapath, one may borrow from Fig. 3/13 (page 3/10).

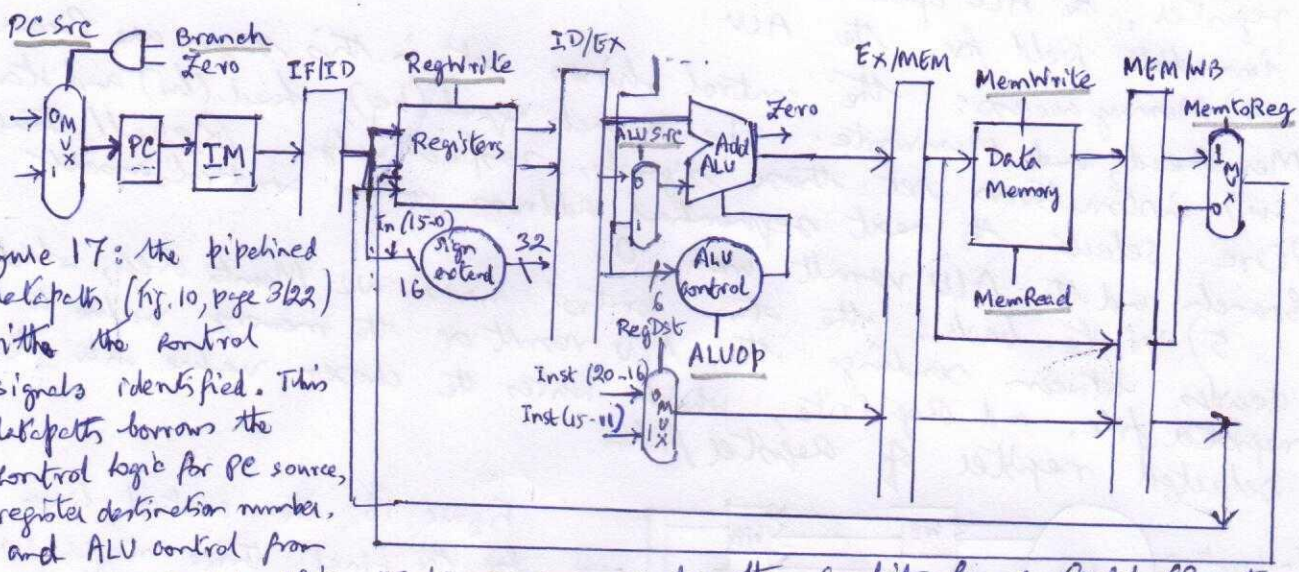


Figure 17: the pipelined datapath (Fig. 10, page 3/22) with the control signals identified. This datapath borrows the control logic for PC source, register destination number, and ALU control from earlier material. Note that now one needs the 6-bit funct field (function code) of the instruction in the EX stage as input to ALU control, so these bits must also be included in the ID/EX pipeline register. Recall that six bits are also the 6 least significant bits of the immediate field in the instruction, so the ID/EX pipeline register can supply them from the immediate field since sign extension leaves these bits unchanged

• Figure 17 shows how the control lines are labeled on the existing pipelined datapath: One borrows (as much as one can) from the control for the datapath shown in Fig. 3/13 (page 3/10). In particular, one uses the same ALU control logic, branch logic, destination-register-number multiplexor, and control lines. These functions are defined in Fig. 10 (page 3/8), Fig. 11 (reverse side of page 3/8) and Fig. 14 (page 3/10)

[Based on 2-bit ALUOp e.g. 00 for LW/SW, 10 for R-type instr and 6-bit funct field, the 4-bit ALU control bits for desired ALU action like add, sub, AND, OR, set on less than are generated]

Note that as PC (program counter), and pipeline registers (IF/ID, ..., MEM/WB) are written during each clock cycle, for them there are no separate write control signals.

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

1) Instruction fetch: the control signals to read the instruction memory and to write the PC are always enabled/asserted, so there is nothing special to control in this pipeline stage.

P.T.O.

2) Instruction decode / register file read: as in the previous stage, same thing happens at every clock cycle, so there exist no optional control lines to set.

3) Execution / address calculation: the signals to be set are RegDst, ALUOp, and ALUSrc. Those signals select the Result register, the ALU operation, and either Read data 2 or a sign-extended immediate field for the ALU

4) Memory access: the control lines set in this stage are Branch, MemRead, and MemWrite. The branch equal (beq), load (lw) and store (sw) instructions set those signals, respectively. Recall that Pcsrc 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 RegWrite, which writes the chosen value into the selected register of Register file

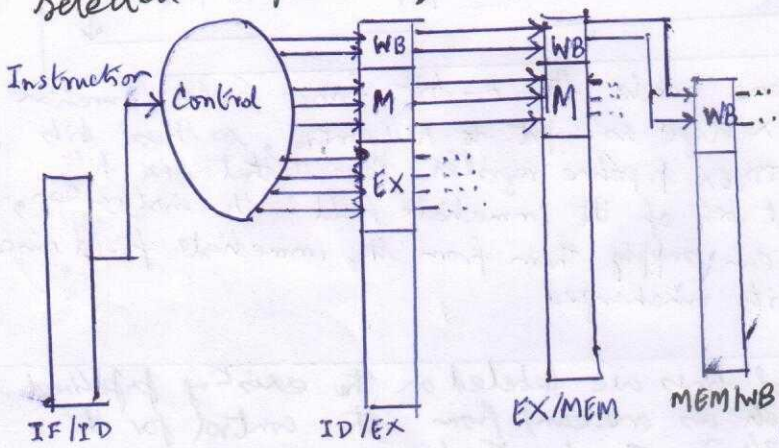


Figure 18: the control lines for the final three stages: note that four of the nine control lines, viz. RegDst, ALUOp1, ALUOp0 and ALUSrc are used in the EX phase, with the remaining five control lines passed on to the EX/MEM pipeline register (which is extended to hold the control lines. Three of these five signals, namely Branch, MemRead and MemWrite are used in MEM stage, while the remaining two, viz RegWrite and MemtoReg will be used in WB stage.

, Since the control lines start with the EX stage, one can create the control information during instruction decode. Figure 18 above shows that these control signals are then used in the appropriate pipeline stage as the instruction moves down the pipeline.

Parallelism via Instructions

• Pipelining exploits the potential parallelism among instructions. This parallelism is called instruction-level parallelism (ILP). Two methods exist to increase the potential amount of ILP. The first is increasing the depth (increasing the no. of stages) of the pipeline to overlap more instructions. With more operations being overlapped, the amount of parallelism being exploited is higher. Performance is potentially greater since the clock cycle can be shorter.

• Another approach is to replicate the internal components of the computer so that it can launch multiple instructions at every pipeline stage. The general name for this technique is multiple issue. Launching multiple instructions per stage allows the instruction execution rate to exceed the clock rate, in other words, the CPI (cycles per instruction) to be less than 1. (or increasing the instructions per clock cycle (IPC in short)). Hence, a 4 GHz four-way multiple-issue microprocessor can execute a peak rate of 16 billion instructions per second and have a best-case CPI of 0.25, or an IPC of 4. Assuming a five-stage pipeline, such a processor would have 20 instructions in execution at any given time. Today's high-end microprocessors attempt to issue from three to six instructions in every clock cycle. Even moderate designs will aim at an IPC of 2. There are however, many constraints like what types of instructions may be executed simultaneously, and what happens when dependencies arise.

• There are two major ways to implement a multiple-issue processor, with the major difference being the division of work between the compiler and the hardware. As the division of work dictates whether decisions are being made statically (i.e. at compile time) or dynamically (i.e. during execution), the approaches are sometimes called static multiple issue and dynamic multiple issue. As will be seen, both approaches have other, more commonly used names, which may be less precise or more restrictive.

Note: 1) static multiple issue: an approach to implementing a multiple-issue processor where many decisions are made by the compiler before execution

2) dynamic multiple issue: an approach to realizing a multiple-issue processor where many decisions are made during execution by the processor.

P.T.O.

. There are two primary and distinct responsibilities that must be dealt with in a multiple-issue pipeline:

1) packaging instructions into issue slots: (note that issue slots are the positions from which instructions could be issued in a given clock cycle) how does the processor determine how many instructions and which instructions can be issued in a given clock cycle? In most static issue processors, this process is at least partially handled by the compiler. In dynamic issue designs, it is normally dealt with at runtime by the processor, although the compiler will often have already tried to help improve the issue rate by placing the instructions in a beneficial order.

2) dealing with data and control hazard: in static issue processors, the compiler handles some or all of the consequences of data and control hazards statically. In contrast, most dynamic issue processors attempt to alleviate at least some classes of hazards using hardware techniques operating at execution time

The concept of Speculation: one of the most important methods for finding and exploiting more ILP is speculation. Based on the idea of prediction, speculation is an approach that allows the compiler or the processor to "guess" about properties of an instruction, so as to enable execution to begin for other instructions that may depend on the speculated instruction. For example, one may speculate on the outcome of a branch, so that instructions after the branch could be executed earlier. Another example is that one may speculate that a store ("sw") that precedes a load ("lw") does not refer to the same address, which would allow the load to be executed before the store