Pipelining : is an implementation technique in which multiple instructions are overlapped in execution.

As all of us are used to deal with laundry (clothes and linen which are washed and ironed on regular basis), consider the following non-pipelined approach to laundry as follows:-

1) place one dirty load of clothes in the washer [W]
2) after the washer job is done, place the wet load on dryer [D]
3) when the dryer is finished, place the dry load on a table and fold. [F]
4) when folding is done, one can store the clothes in a wardrobe [S].

When the above load is completed, start over with the next dirty load. The pipelined approach takes much less times as figure 1 shows

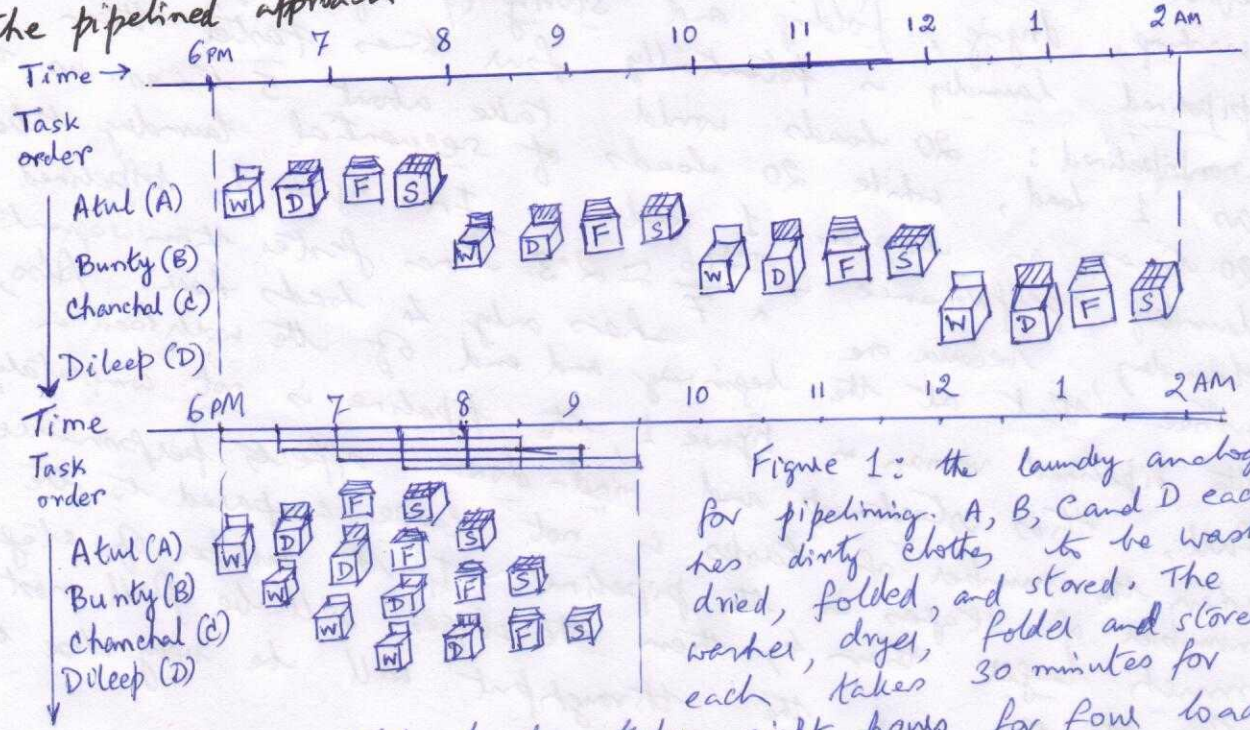


Figure 1: the laundry analogy for pipelining. A, B, C and D each has dirty clothes to be washed, dried, folded, and stored. The washer, dryer, folder and storer each takes 30 minutes for their task. Sequential laundry takes eight hours for four loads of wash, while pipelined laundry takes just 3.5 hours. One shows the pipeline stage of different loads over time by showing copies of four resources on this two-dimensional line, but one really has just one of each resource.

- As shown above, the pipelined approch takes much less time. As soon as the washer is finished with the first load, which is placed in the dryer, one can load the washer with second dirty load. When the first load is dry, one can place it on the table to start folding, move the wet load to the dryer, and put the next dirty load into the washer. Next,

P.T.O.

the first load can be put on the stove, the second load can be folded, the dryer can have the third load, and the fourth load can be on the washer. At this point, all steps — called stages in pipelining — are operating concurrently. As long as one has separate resources for each stage, one can pipeline the tasks.

Note: pipelining would not decrease the time to complete one load of laundry. The reason pipelining is faster for many loads is that several stages work in parallel, and so more loads are finished per hour. (With many loads of laundry to do, the improvement in throughput decreases total time to complete the work.

. If all stages take about same amount of time, and there is enough work to do, then the speed-up due to pipelining is equal to the number of stages in pipeline (in this case four: washing, drying, folding and storing (putting away)). Therefore pipelined laundry is potentially four times faster than nonpipelined: 20 loads would take about 5 times as long as 1 load, while 20 loads of sequential laundry takes 20 times as long as 1 load. In figure 1, pipelined laundry performance is only $\frac{16}{7} \simeq 2.3$ times faster than sequential laundry, because one has only 4 loads here. Also, notice that at the beginning and end of the workload in the pipelined version in figure 1, the pipeline is not completely full; this start-up and wind-down affects performance when the number of tasks is not large compared to the number of stages in the pipeline. If the number of stages is much larger than 4, then the stages will be full most of the time, and the throughput will be very close to 4!

Pipelining (cont.)

- The same principles apply to processors where one pipelines execution of instructions. Five steps for MIPS instructions are as follows:

1) Fetch instruction from memory.
2) Read registers while decoding the instruction. The regular format of MIPS instructions allows reading and decoding to occur at the same time
3) Execute the operation or calculate an address.
4) Access an operand in data memory.
5) Write the result into a register.

<Thus, the MIPS pipeline one wishes to exploit has five stages. The following example establishes that pipelining speeds up instruction execution (as it speeds up the laundry)>

Single-Cycle versus Pipelined Performance

- Let us limit our attention to eight instructions, namely load word (lw), store word (sw), add (add), subtract (sub), AND (and), OR (or), set less than (slt), and branch on equal (beq).

Compare the average time between instructions of a single-cycle implementation, in which all instructions take one clock cycle, to a pipelined implementation. The operation times for the major functional units in this example are 200 ps for memory access, 200 ps for ALU operation, and 100 ps for register file read or write. In the single-cycle model, every instruction takes exactly one clock cycle, so the clock cycle must be stretched to accommodate the slowest instruction.

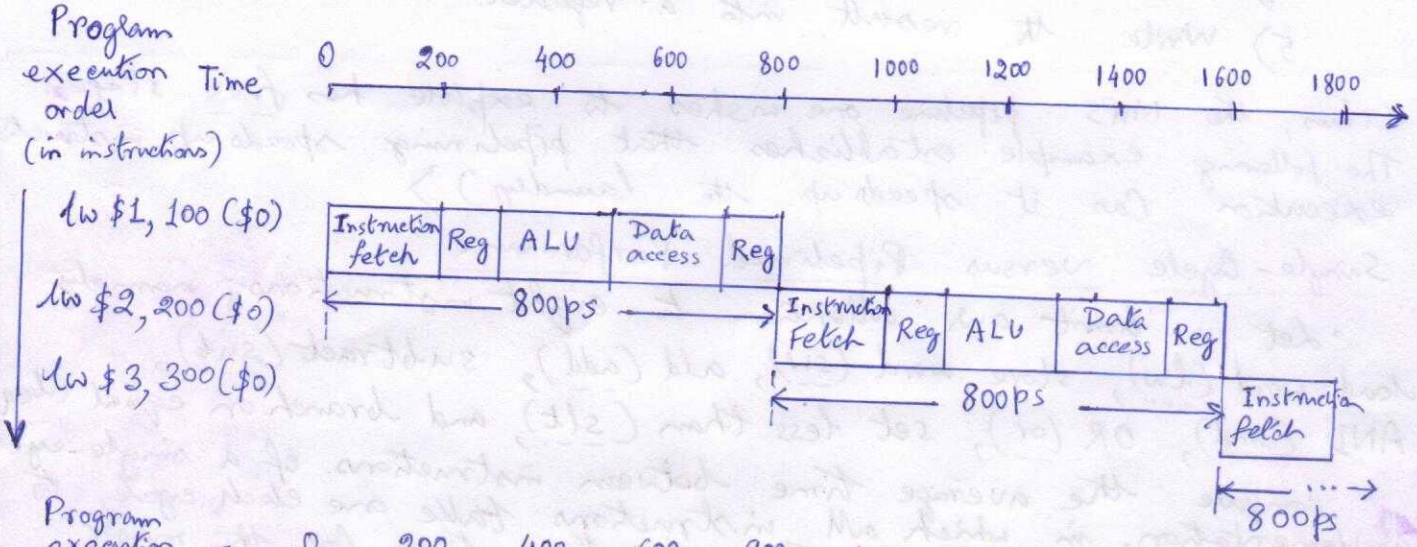
- Figure 2 (shown overleaf) shows the time required for each of the eight instructions. The single-cycle design must allow for the slowest instruction (in Fig. 2 it is lw — so the time required for every instruction is 800 ps). Similarly to Figure 1, Figure 3 compares nonpipelined and pipelined execution of three load word instructions. Thus, the time between the first and the fourth instructions in the nonpipelined design is 3 × 800 ps or 2400 ps.

- All the pipeline stages take a single clock cycle, so the clock cycle must be long enough to accommodate the slowest operation. Just as the single-cycle design must take the worst-case clock cycle of 800 ps, (even though some instructions can be as fast as 500 ps) the pipelined execution clock cycle must have the worst-case clock cycle of 200 ps, (even though some stages take only 100 ps).

P.T.O.

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (lw)	200ps	100ps	200ps	200ps	100ps	800ps
Store word (sw)	200ps	100ps	200ps	200ps		700ps
R-format (add, sub, AND, OR, slt)	200ps	100ps	200ps		100ps	600ps
Branch (beq)	200ps	100ps	200ps			500ps

Figure 2: Total time for each instruction calculated from the time for each component. (this calculation assumes that the multiplexors, control unit, PC accesses, and sign extension unit have no delay)



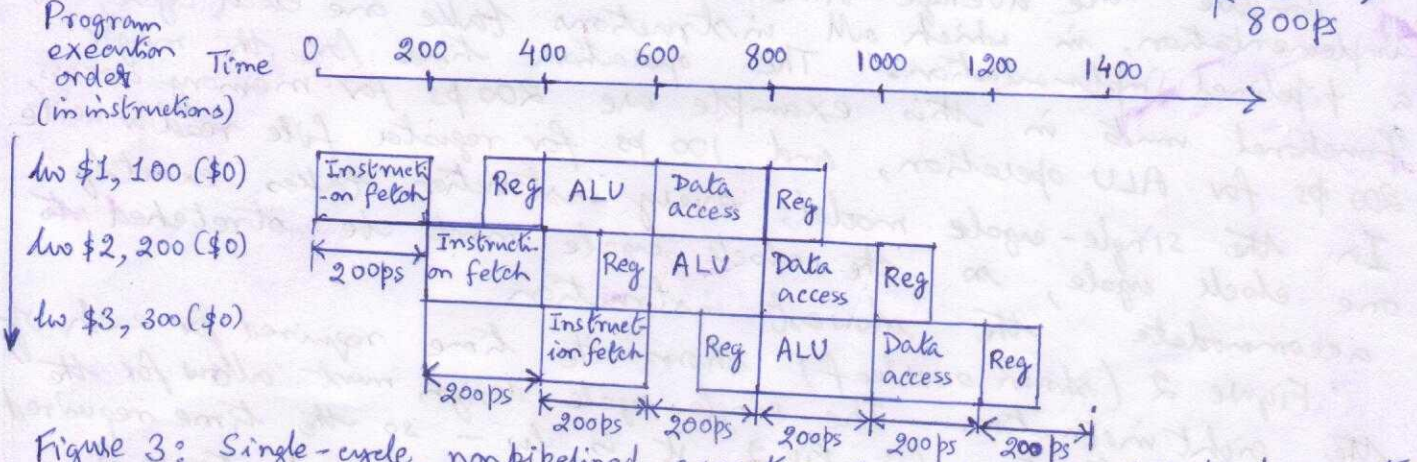


Figure 3: Single-cycle, nonpipelined execution in top versus pipelined execution in bottom. Both use the same hardware components, whose time is listed in Fig 2. In this case, one can see a four fold speed-up on average time between instructions, from 800 ps down to 200 ps. <One can compare this figure to Fig. 1. For the laundry, all stages were assumed to be equal. If the dryer were the slowest then the dryer stage would set the stage time. The pipeline stage times of a computer are also limited by the slowest resource, either the ALU operation or the memory access.> One assumes the write to the register file occurs in the first half of the clock cycle and the read from the register file occurs in the second half.

Pipelining Overview (Cont.)

Pipelining still offers a fourfold performance improvement: the time between the first and fourth instructions is 3×200 ps or 600 ps.

Let us derive a formula out of the above discussion on pipelining. If the stages are perfectly balanced, then the time between instructions on the pipelined processor – assuming ideal conditions – is equal to.

$$\text{Time between instructions}_{\text{pipelined}} = \frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of pipe stages}}$$

Under ideal conditions and with a large number of instructions, the speed-up from pipelining is approximately equal to the number of pipe stages; 5-stage pipeline is nearly 5 times faster. As suggested by the formula, a 5-stage pipeline should offer nearly a fivefold improvement over the 800 ps nonpipelined time, or a 160 ps clock cycle. This example however shows that the stages may not be balanced properly. Also, as some overhead is involved in pipelining, the time per instruction in the pipelined processor will exceed the minimum possible one, and speed-up will be less than the no. of pipeline stages.

Moreover, the claim of 4-fold improvement in this example is not reflected in the total execution time for the three instructions. For three instructions, (very small no. of inst^{n}s)

$$\text{speedup} = \frac{\text{tot. exec. time in nonpipelined case}}{\text{" " " pipelined case}} = \frac{2400 \text{ ps}}{1400 \text{ ps}}$$

If one adds 1,000,000 (10^6) instructions in the pipelined example, each inst^{n} adds 200 ps to the total execution time, while in non-pipelined case, it adds 800 ps for each inst^{n}.

$$\text{So, speedup} = \frac{1{,}000{,}000 \times 800 + 2400}{1{,}000{,}000 \times 200 + 1400} = \frac{800{,}002{,}400}{200{,}001{,}400}$$

$$\simeq \frac{800}{200} \text{ (neglecting the small parts both in num. and denominator)} \simeq 4.00$$

- Pipelining improves performance by increasing instruction throughput, as opposed to decreasing the execution time of an individual instruction (as real programs execute billions of instructions, instruction throughput is important metric).

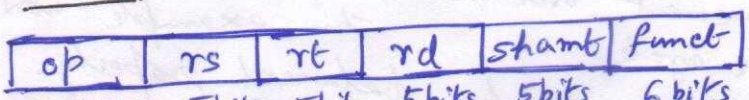
P.T.O.

Designing Instruction Sets for Pipelining.

Note that MIPS instruction set was designed for pipelined execution.

- First, all MIPS inst^{n}s are of same length. This restriction makes it much easier to fetch instructions in the first pipeline stage and to decode them in the second stage. In an instruction set like the x86, where instructions vary from 1 byte to 15 bytes, pipelining is much more challenging. < Modern implementations of the x86 architecture actually translate x86 instructions into simple operations that look like MIPS instructions, and then pipeline the simple operations rather than the native x86 instructions. >

- Secondly, MIPS has only a few instruction formats, with the source register fields being located in the same place in each instruction. This symmetry means that the second stage can begin reading the register file at the same time that the hardware is determining what type of instruction was fetched. If MIPS instruction formats were not symmetric, one would need to split stage 2, resulting in six pipeline stages.



- Third, memory operands only appear in loads or stores in MIPS. This restriction means one can use the execute stage (third stage) to calculate the memory address and then access memory in the next stage. < If one could operate on the operands in memory, as in the x86, stages 3 and 4 would expand to an address stage, memory stage, and then execute stage >

- Fourth, operands must be aligned in memory. Hence, one need not worry about a single data transfer instruction requiring two data memory accesses; the requested data can be transferred between processor and memory in a single pipeline stage.

Pipeline Hazards

There exist situations in pipelining when the next instruction cannot execute in the following clock cycle. These events are called hazards.

Structural hazard: hardware cannot support the combination of instructions that one wishes to execute in the same clock cycle. Suppose one had a single memory instead of two memories. (i.e. no separate inst^n mem. and data memory). If the pipeline in fig. 3 had a fourth instruction, then in the same clock cycle (t_c = 600ps around to 800 ps), the first inst^n is accessing data from memory while the fourth inst^n is fetching an inst^n from the same mem (i.e IM and DM are same as assumed). Thus without two memory modules, the pipeline could have a structural hazard.

Data Hazards: occur when the pipeline must be stalled because one step must wait for another to complete. In a computer pipeline, data hazards arise from the dependence of one instruction on an earlier one that is still in the pipeline. For example, suppose one has an add instruction followed immediately by a subtraction instruction that uses the sum ($s0):

```
add $s0, $t0, $t1
sub $t2, $s0, $t3
```

Without intervention, a data hazard can severely stall the pipeline. The add instruction does not write its result until the fifth stage, which means that one has to waste three clock cycles in the pipeline. This is because $s0 is the destination register in first (add) instruction, while it is the source register in the second (sub) instruction.

These dependences happen so often and the delay is so long that one cannot rely (and depend) on the compiler to remove all such hazards. The primary solution is based on observation that one does not need to wait for the instruction to complete before trying to resolve the data hazard. For the above

P.T.O.

code sequence, as soon as the ALU creates the sum for the add, one can supply it as an input for the subtract. Adding extra hardware to retrieve the missing item early from the internal resources is called forwarding or bypassing.

Example on Forwarding with Two Instructions

To show which pipeline stages would be connected by forwarding Represent the datapaths during the five stages of the pipeline. Align a copy of the datapaths for each instruction.

Sol:

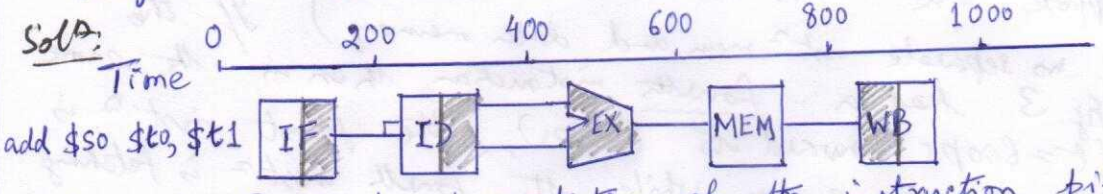


Figure 4: Graphical representation of the instruction pipeline. Here one uses symbols representing the physical resources with abbreviation for pipeline stages. The symbols for the five stages: IF for the instruction fetch stage, (with the box representing instruction memory); ID for the instruction decode / register file read stage (with the drawing showing the register file being read); EX for the execution stage (with the drawing representing the ALU); MEM for the memory access stage (the box representing the data memory); and WB for the write-back stage (with the drawing showing the register file being written). The shading indicates the element being used by the instruction. Hence, MEM has a white background because add does not access the data memory. The shading on the right half of the register file or memory means the element is read in that stage, and shading on the left half means it is written in that stage. Hence, the ~~state~~ right half of ID is shaded in the second stage because the register file is read, and the left half of WB is shaded on the left side in the fifth stage because the register file is written.

- In this graphical representation of events, forwarding paths are valid only if the destination stage is later in time than the source stage. For instance, there cannot be a valid forwarding path from the output of the memory access stage in the first instruction to the input of the execution stage of the following (instruction) as that implies going backward in time.

Forwarding with Two Instructions (Cont.)

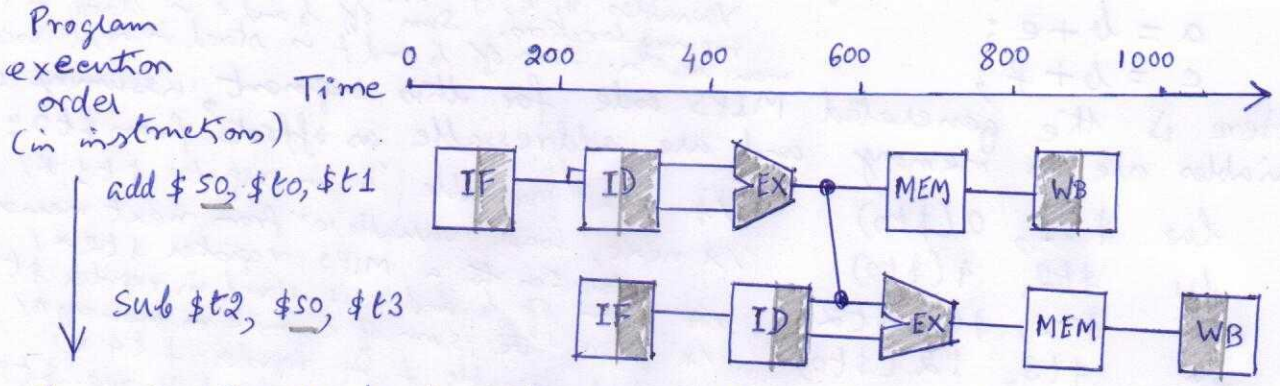


Figure 5: Graphical representation of forwarding: the connection shows the forwarding path from the output of the EX stage of add to the input of the EX stage for sub, replacing the value from register $s0 read in the second stage of sub

- Though forwarding works very well, it cannot prevent all pipeline stalls. For example, suppose the first instruction was a load of $s0 instead of an add. As ~~only~~ one can imagine (from looking at Fig. 5) the desired data will be available only after the fourth stage of the first instruction in the dependence, which is too late for the input of the third stage of sub. Hence, even with forwarding ~~one after forwarding~~ one would have to stall one stage for a load-use data hazard, as Fig. 6 shows. This figure shows an important pipeline concept, called formally a pipeline stall, but often given the nickname bubble

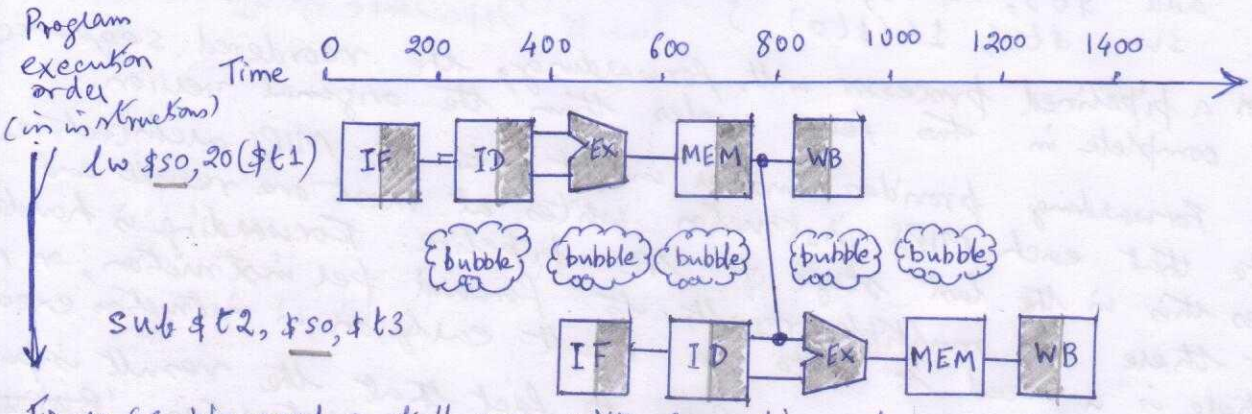


Figure 6: We need a stall even with forwarding when an R-format instruction following a load tries to use the data. Without the stall, the path from memory access stage output to execution stage input would be going backward in time, which is impossible. This figure is actually a simplification, since we cannot know until after the subtract instruction is fetched and decoded whether or not a stall will be necessary.

Example: Reordering Code to Avoid Pipeline Stalls

Consider the following code segment in C:

```
a = b + e;
c = b + f;
```

Variables b, e, f are stored in three consecutive memory locations. Sum of b and e is stored in next mem. location. Sum of b and f is stored in next location.

Here is the generated MIPS code for this segment, assuming all variables are in memory and are addressable as offsets from $t0:

```
lw   $t1, 0($t0)     /* load variable 'b' in register $t1 */
lw   $t2, 4($t0)     /* next, load variable 'e' from next memory
                        location to a MIPS register $t2 */
add  $t3, $t1, $t2   /* sum of b and e is stored in register $t3 */
sw   $t3, 12($t0)    /* store the sum of b and e in memory */
lw   $t4, 8($t0)     /* load variable f in register $t4 */
add  $t5, $t1, $t4   /* sum of b and f is stored in reg. $t5 */
sw   $t5, 16($t0)    /* store contents of reg. $t5 in next mem. location
```

Find the hazards in the preceding code segment and reorder the instructions to avoid any pipeline stalls.

Soln: Both add instructions have a hazard because of their respective dependence on the immediately preceding 'lw' instructions. Note that bypassing / forwarding eliminates several other potential hazards, including the dependence of the first add on the first lw and any hazards for store instructions. Moving up the third lw instruction to become the third instruction (of the code sequence) eliminates both hazards:

```
    lw   $t1, 0($t0)
    lw   $t2, 4($t0)
→   lw   $t4, 8($t0)
    add  $t3, $t1, $t2
    sw   $t3, 12($t0)
    add  $t5, $t1, $t4
    sw   $t5, 16($t0)
```

does data hazard exist here, as reg. $t3 must be written before it can be stored in memory?

Similarly, does data hazard exist due to dependence of sw on add?

- On a pipelined processor with forwarding, the reordered sequence will complete in two fewer cycles than the original version.

N.B. Forwarding provides another insight into the MIPS architecture. Note that each MIPS instruction writes at most one result and does this in the last stage of the pipeline. Forwarding is handled if there are multiple results to forward per instruction, or if there is a need to write a result early on in instruction execution.

The name 'forwarding' comes from the fact that the result is passed forward from an earlier instruction to a later instruction. 'Bypassing' comes from passing the result around the register file to the desired unit, e.g. ALU input.

Control Hazards

• A control hazard arises from the need to make a decision based on the results of one instruction while others are executing. It is also called branch hazard. When the proper instruction cannot execute in the proper pipeline clock cycle because the instruction that was fetched is not the one that is needed; that is, the flow of instruction addresses is not what the pipeline expected.

• First solution is to stall, the control hazard faced during branch instruction is simply to stall ~~just operate repeatedly~~. Note that we must begin fetching the instruction following the branch on the very next clock cycle. However, the pipeline cannot possibly know what the next instruction should be, since it only just received the branch instruction from memory. One possible solution is to stall immediately after we fetch a branch (instruction), waiting until the pipeline determines the outcome of the branch and knows what instruction address to fetch from.

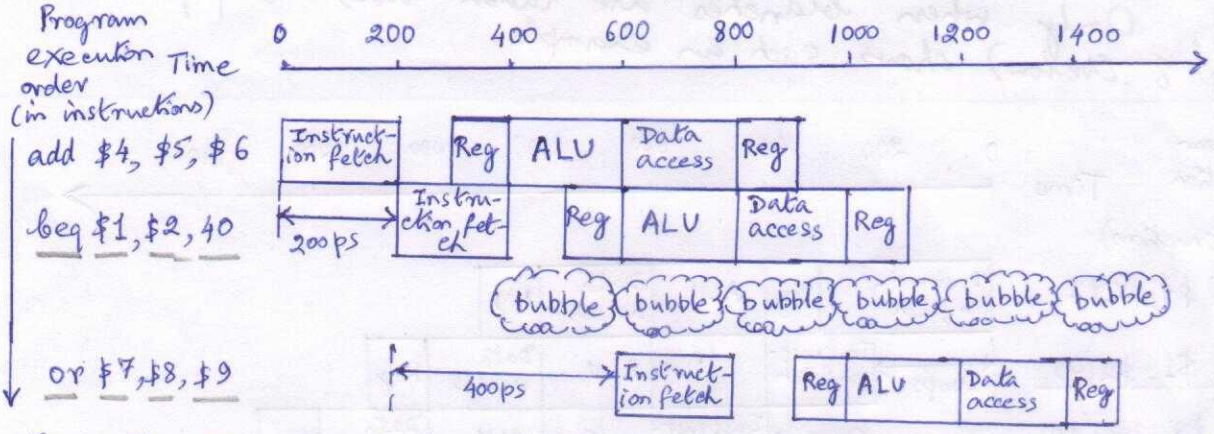


Figure 7: Pipeline showing stalling on every conditional branch as solution to control hazards: this example assumes that the conditional branch is taken, and the instruction at the destination of the branch is the OR instruction. There is a one-stage pipeline stall, or bubble, after the branch.

• Let us assume that we employ enough extra hardware so that we can test registers, calculate the branch address, and update the PC during the second stage of the pipeline. Even with this additional hardware, the pipeline involving conditional branches would look like Fig. 7 above. The or instruction, executed if the branch fails, is stalled one extra 200 ps clock cycle before starting.

Example on Performance of "stall on Branch"

To estimate the impact on the clock cycles per instruction (CPI) of stalling on branches. Assume all other instructions have a CPI of 1.

Answer: It is shown that branches are 17% of the instructions executed in SPECint2006 benchmark. Since the other instructions run have a CPI of 1, and branches took one extra clock cycle for the stall, then one would see a CPI of 1.17 and hence a slowdown of 1.17 versus the ideal case.

- If one cannot resolve the branch in the second stage, as is often the case for longer pipelines, then one would see an even larger slowdown if one stalls on branches. The cost of this (stalling) option is too high for most computers to use. This motivates a second solution to the control hazard problem, which is based on using prediction.
- Computers do indeed use prediction to handle branches. One simple approach is to predict always that branches will be untaken (i.e. not taken). When one is right, the pipeline proceeds at full speed. Only when branches are taken does the pipeline stall. Figure 8 (below) shows such an example.

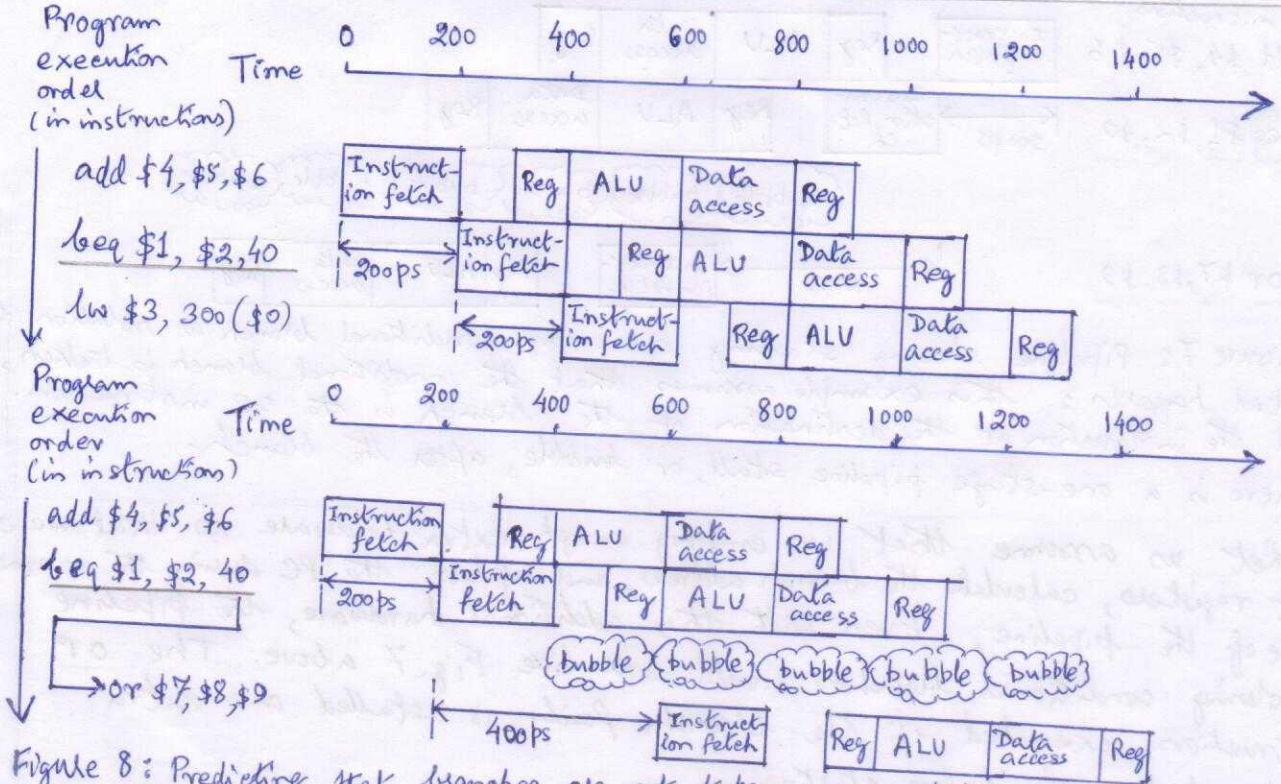


Figure 8: Predicting that branches are not taken as a solution to control hazard. The top drawing shows the pipeline when the branch is not taken. The bottom drawing shows the pipeline when the branch is taken. As noted in Figure 7, the insertion of a bubble in this manner simplifies what actually happens at least during the first clock cycle immediately following the branch.

Control Hazards (Cont.)

- A more sophisticated version of branch prediction (which is a method of resolving a branch hazard, that assumes a given outcome for the branch and proceeds from that assumption rather than waiting to ascertain the actual outcome). would have some branches predicted as taken and some as untaken Namely in programming, at the bottom loops are branches that jump back to the top of the loop. Since they are likely to be taken and they branch backward, one can always predict taken for branches that jump to an earlier addres.

Such rigid approaches to branch prediction rely on stereotypical behaviour, and do not account for the individuality of a specific branch instruction. Dynamic hardware predictors, on the other hand, make their guesses depending on the behavior of each branch, and may change predictions for a branch over the life of a program.

A popular approach to dynamic prediction of branches: to keep a history for each branch as taken or untaken, and then use the recent past behavior to predict the future. As may be seen later, the amount and type of history kept have become extensive, with the result being that dynamic branch predictors can correctly predict branches with more than 90% accuracy. When the guess is wrong, the pipeline control must ensure that the instructions following the wrongly guessed branch have no effect and must restart the pipeline from the proper branch address.

As in the case of all other solutions to control hazards, longer pipelines worsen the problem, in this case by raising the cost of misprediction.

- Third approach to mitigate control hazard is delayed branching, which is the solution actually used by the MIPS architecture. This procedure always executes the next sequential instruction, thus the branch taking place (if it does at all) after delay of executing that instruction. It is hidden from the MIPS assembly language programmer, because the assembler can automatically arrange the instructions to get the branch behaviour desired by the programmer.

MIPS software will place an instruction immediately after the delayed branch instruction that is not affected by the branch, and in case the branch is taken, the ~~sftw~~ system software will change the address of the instruction that follows this safe instruction. In the present example, the 'add' instruction before the 'branch' inst^{n} does not affect the branch, and can be moved after the branch to fully hide the branch delay. Since delayed branches are useful when the branches are short, so processor uses a delayed branch of more than one cycle. For longer branch delays, a hardware-based branch prediction (which will be quicker?) is usually adopted.

```
add $4, $5, $6
beq $1, $2, 40   <- in case branch is taken, or
  -> or $7, $8, $9   inst'n will be executed (at the destination of the branch)
```

Summary of Overview on Pipelining

- Pipelining is a technique that exploits parallelism among the instructions in a sequential instruction stream. It has considerable advantage that, unlike programming a multiprocessor, it is basically invisible to the programmer.
- One can now examine the design of a pipelined datapath and basic control to understand how pipelining is implemented, and the challenges of dealing with hazards. Using this understanding, one can next explore implementation of forwarding and stalls. Subsequently, one can learn about solutions to branch hazards, and then understand how exceptions are handled.
- Beyond the memory system, the effective operation of the pipeline is usually the most important factor in determining the CPI of the processor and hence its performance. It is quite complex to understand the performance of a modern multiple-issue pipelined processor. However, structural, data, and control hazards remain important in both simple pipelines and more sophisticated ones.
- For modern pipelines, the floating-point unit is more likely to be affected by structural hazard. Whereas, control hazards affect the integer programs more, as these programs have higher branch frequencies as well as less predictable branches. Data hazards can pose performance bottlenecks in both integer and floating-point programs. Often it is easier to deal with data hazards in floating-point programs because the lower branch frequency and more regular memory access patterns allow the compiler to try to schedule instructions to avoid hazards. It is more difficult to perform such optimizations in integer programs that have less regular memory access, involving more use of pointers.