Homework 2 Question 3: Computer Pipelining

CIS-655 ADVANCED COMPUTER ARCHITECTURE

PROF. MOHAMMED ABDALLAH

10/15/2021

Anthony Redamonti
SYRACUSE UNIVERSITY

Introduction to Computer Pipelining

Have you ever been to a popular car wash? A car enters the first stage of the car wash and is drenched in soapy water. Then it will move onto the next stage of being scrubbed down by large sponges. Then it will be washed with clean water and dried. Thus, there are four stages to this car wash. When one car exits a stage in the car wash, another will begin that same stage. The car wash will be able to wash the maximum possible number of cars by operating in this fashion. Computer pipelining is very similar in concept.

Computer pipelining improves the throughput of the system by breaking instructions into smaller sub-tasks to be performed by the CPU. Typically, instructions can be broken into five sub-tasks: Instruction Fetch, Instruction Decode/Operand Fetch, Instruction Execute, Memory Read/Write and Writeback. Like the car wash model, if the CPU is performing one of the sub-tasks of an instruction, it can simultaneously perform a different sub-task of a different instruction, like washing car 2 while drying car 1.

Pipelined Architecture

In a pipelined CPU, the hardware can be divided into segments that perform the various sub-tasks. These segments are connected in series so that the output of one segment is the input of the next. The number of segments varies between processors. These segments are also called “stages.” There are control units that control all stages using control signals. Each stage has a set of internal registers to store its output data. These registers act as input data for the next stage. The global clock is used to synchronize the flow of data between all stages. The clock signal must pass through all stages in the pipeline before the next clock signal can commence.

As previously mentioned, the stages are:

1) **Instruction Fetch (IF)**: The first stage simply fetches the instruction to be executed. The instruction is taken from the main memory and stored in a set of registers called “Instruction Registers” (IR). The length of the instruction depends on the size of the computer architecture (64-bit, 32-bit, etc.).
2) **Instruction Decode/Operand Fetch (ID/OF)**: The bits of the instruction are decoded. The opcode is read along with the appropriate operands. Operands are the registers or addresses that will be needed to perform the instruction. The bits are converted to their appropriate control signals and fed into the registers of the next stage.
3) **Instruction Execute (EX)**: The instruction is executed. The ALU carries out the calculation using the control signals through the appropriate logic circuits. If a branch or load/store operation is commanded, the address for the command is gathered.
4) **Memory Read/Write (MEM)**: If the operation is to read or write to memory (load/store) then the memory is accessed during this step. The read/write operation is executed.
5) **Write Back (WB)**: the result of the operation is written back to the appropriate register.

There are typically five stages or segments in the multicycle path. In a pipelined system of this type, the processor can execute a maximum of five (or N number of) instructions per clock cycle. For its counterpart, the non-pipelined system, it would take five (or N number of) clock cycles to complete just one instruction. Thus, it can be said that a pipelined system has a throughput that is N times faster than

a non-pipelined system. However, it is not always the case as there are several events that can cause performance delays in a pipelined system, but it is sufficient to say that on average, the improvement is in the order of N (number of stages).

CISC vs RISC Pipelining

A Reduced Instruction Set Computer (RISC) system is more conductive toward instruction pipelining than its predecessor, the Complex Instruction Set Computer (CISC). Implementing instruction pipelining in a CISC system was complicated and difficult because instructions had sub-tasks of varying length. The RISC system typically has the five sub-tasks, each only taking one clock cycle. In an ideal world, because each sub-task can be completed in only one clock cycle, the Instructions Per Cycle (IPC) averages to 1.

Pipelining in a RISC system helps to reduce the complexity of the computer architecture. The RISC pipeline works to maintain spatial and temporal locality by storing the most frequently used operands in the general-purpose registers. It can support many CPU registers so that the use of the memory bus is decreased (more operands are able to be stored in these registers). In a RISC system, all superfluous and complex code is removed, and only the essential instructions are performed.

Pipelining MIPS Instructions

The MIPS instructions are effective in a RISC system because they're all the same length. Because they are the same length, the five-stage model will be simpler, less prone to error, and easier to implement. There are also few command formats, which decreases hardware complexity. Similarly, the five-bit long fields related to the destination register, source register, and second source register should be represented using the same bits for different command types (R type and I type). These registers must be read at the same time (in the same clock cycle). The only commands in the MIPS instruction set that access memory locations are the load word, *lw*, and store word, *sw*, instructions.

The data path of a system using the MIPS instruction set can be divided into the five stages previously described. Note that there are buffers between each stage in the system. The first stage represents the instruction fetch and contains the program counter which stores the address of the current instruction. The address is fed into the instruction memory, which fetches the operands used in the instruction. The program counter is also fed into an ALU which adds four to the address. The value is sent to a multiplexer where it is decided whether to use this value or another value dedicated to jump or branch instructions.

In the second stage, the buffered operand values are fed into the register file, where their current values are stored. These values along with a sign extension are fed into the buffer for the third stage. The third stage is where the instruction execution occurs and contains two arithmetic logic units (ALU's). The first ALU is used to perform the calculation of the operation and another for adding an offset to the jump address in case of a jump command. The fourth stage is where the memory is stored as well as a mutex used to select the address for the next instruction. Finally, the fifth and last stage contains a mutex that writes data back to the second stage in the register file location.

One important note is that each stage contains control hardware that are connected between stages. In the first stage, there is a control unit which is fed the instruction. It decodes this instruction and feeds data related to each of the next four stages into the next stage's control lines. The second stage (Instruction Fetch) feeds the register operands as well as the opcode of the instruction to the next stage

(Instruction Decode/Execute). The control unit of the second stage will decide which operands to feed into the ALU, which arithmetic operation to perform, and whether to branch to a new address. The third stage's control unit decides if and where to read or write to memory. The control unit for the last stage writes data back to the register file.

Arithmetic Pipelining

Arithmetic pipelining allows for the performance of calculations involving fixed and floating-point integer values. The floating-point numbers are broken up into binary form shown below.

$$X = A * 2^a$$

$$Y = B * 2^b$$

The upper-case A and B represent the mantissa, or the main digits of the number. The lower-case a and b represent the exponents and signify where the decimal point should be placed. Operations such as addition and subtraction can be broken down into four sub-tasks:

1) Compare the exponents.
2) Align the mantissa.
3) Add or subtract the mantissa.
4) Return the result.

In step 1, the exponents are fed into a general-purpose register. The exponents are compared using subtraction, and the result of this operation is used during the aligning of the mantissas. The exponent of greater value is used in the return value, and the difference between the exponents signifies the number of right-shifts needed to be performed on the lower value mantissa. Then the exponent is chosen, and the mantissas are added or subtracted. The results from the mantissa calculation are normalized and the exponent in adjusted. Finally, the results are fed into two general-purpose registers. Floating point operations typically inject a total of only three cycles of latency, whereas other operations are typically executed in one cycle.

Code Reordering

A CPU may not be able to perform instruction pipelining if there are dependencies between consecutive instructions or if there is branching. Thus, in the real world the IPC is typically greater than 1. An example is shown below.

```
Sub $s1, $s2, $s3     # register s1 = s2 – s3
Addi $t0, $s1, 100    # register t0 = s1 + 100
```

The CPU can perform pipelining up until the operand fetch of the second instruction. Because the operand register $s1 is going to change value in the first instruction, the second instruction must wait for the first instruction to complete before continuing to its operand fetch sub-task. These delays are known as bubbles and greatly affect systems with longer pipelines as the CPU must wait for all the sub-tasks to be complete before continuing.

The solution to this problem is *code reordering*. If there are dependencies between instructions, the code can be reordered so that the CPU can simultaneously execute non-dependent instructions. The

code reordering is generally performed by the compiler and lowers the IPC of the system. A *hardware-dependent compiler* is one which has the capability to reorder code. Typically, the compiler will reorganize code, or the assembler will reorganize object code. The compiler must be specially programmed to synchronize with the timing of the system clock in this case.

Result Forwarding Logic

Another solution to resolving instruction dependencies is to use *result forwarding logic*. If an instruction requires a register value that is already stored inside of an internal register, the CPU does not need to load the register value from memory but can instead use the value in the internal register. The process typically saves one clock cycle per "forward." It allows all instructions that are dependent on a single instruction to be grouped together and simultaneously fed the dependent data via an internal register.

Control Hazards: Branch Prediction

As previously stated, branching is another source of lowering system efficiency. See the example below.

```
Loop:
Sub $s1, $s2, $s3        # register s1 = s2 – s3
Addi $s3, $s1, 1         # register s3 = s1 + 1
Beq $s1, $s3, Loop       # if register s1 equals s3, repeat the loop.
```

Because there are dependencies in the loop between registers s1 and s3, the CPU must wait for the first two instructions to finish before executing the branch instruction. However, to improve system efficiently, the CPU can predict or guess the outcome of the branch instruction. In a loop, some CPU's use a unary prediction method where the result of the previous branch instruction is remembered and chosen as the prediction for the next branch instruction.

However, the system efficiency is lowered as the system will make more incorrect guesses. In this case, the results of all ongoing instructions must be cleared (erased) and started over. To reduce the number of incorrect predictions, many systems employ a binary guessing method. If the previous two branch instructions resulted in executing the loop, then the CPU will choose this as its prediction. Once an incorrect guess occurs, the ongoing tasks are erased and restarted, and branch prediction is not used. Only when two consecutive branch operations have matching results will branch prediction begin again. Other processors have the capability to predict both scenarios (branch back to loop and break the loop).

The process previously described is called *static scheduling*. It involves the compiler making these branch predictions. The compiler will have to either: execute an instruction that is independent of the loop, choose to continue the loop and continue execution, or choose to break the loop and continue execution. The other method of scheduling is *dynamic scheduling* and takes place when the CPU chooses the next instruction to execute at run-time.

Another solution that is commonly implemented is to reduce the latency of the branch instruction. Computer engineers will add additional circuitry to the instruction decode stage so that the branch calculation will take place one cycle sooner. Before, the branch calculation takes place in the instruction

execution stage of the system. Branch prediction (or guessing), as discussed in the previous three paragraphs, may result in a branch error (incorrect guess). If an incorrect guess was made, the data stored in the first three stages would need to be flushed. However, if the branch calculation is performed one cycle sooner, then only the first two stages need to be flushed in the event of an incorrect guess.

Structural Hazards

Structural hazards arise when one instruction needs to access a memory location that is already being used by another instruction. If the first instruction needs to read a memory location and the second instruction needs to write to that same location, how would one satisfy both operations in a timely manner? One could create a duplicate (copy) of the register value for other instructions to access but doing so would create data invalidation issues. The data in one operation would no longer be valid if it was changed simultaneously by another operation.

Instead, computer engineers will allow access to a memory location by splitting the clock cycle in half. The first half of the cycle is given to the first instruction, and the second half is given to the second instruction. There is usually a very small delay in between the two operations dividing them inside one clock cycle. Note that this solution requires very expensive hardware. Another solution would be to add two cache memory banks that either instruction could access. It is more common for engineers to solve structural hazards using hardware that is already on the chip. The second solution is not only more cost effective, but easier to implement.

Other structural hazards could arise from not having the hardware necessary to perform branch instructions if only one ALU was in use. The current MIPS architecture uses two ALU's: one to perform the branching address calculation and one to perform a subtraction operation to determine if the branching condition has been met. If there were only one ALU, it would need two adders to perform both tasks in the same clock cycle.

Advanced Pipelines

There are more advanced versions of pipelining. Compare the standard template previously discussed to a system without pipelining. In a system without instruction pipelining, each instruction is finished before the next instruction begins. The standard pipelining template using five sub-tasks allows the CPU to execute five instructions simultaneously, which makes the system throughput five times faster on average. In more advanced systems, the number of sub-tasks is increased by breaking the *instruction fetch* and *writeback* sub-tasks each into two halves and by adding sub-tasks. The system throughput is therefore increased by a factor of the number of sub-tasks created, assuming each sub-task takes the same amount of time to complete.

Conclusion

Computer pipelining is essential to increase the throughput of the system. In a pipelined CPU, the hardware can be divided into segments that perform the various sub-tasks. Pipelining in a RISC system helps to reduce the complexity of the computer architecture. Issues related to data dependency, hardware resources, control and structural hazards can be mitigated using techniques such as branch prediction, result forwarding logic, and code reordering. As computer architecture evolves, pipelining will always be of great importance in system design.