Architectural Innovations Review

Cache Memory

Small, fast memory for faster program execution

Works because of principle of locality of reference

Contributes directly to increased execution speed, as long as cache hit ratios are high

ECC

Increases system reliability

Goal is reliability not improved execution speed

Virtual Memory

Increases system usability, allowing large process images to spill over from RAM onto hard drive

Does not directly contribute to faster execution performance. In fact, there is a small performance decrease, but the benefit to usability far outweighs the small performance penalty.

Without VM, a process would have to completely fit in RAM in order to execute. Depending on how much RAM was available at a specific time, some processes would simply not be allowed to run.

IEEE 754

Standard for improving the accuracy of floating point calculations

Execution speed not a major goal

Instruction Pipelining

Hardware innovation to speed up the von Neuman instruction execution cycle

Produces significant execution speedup

Instruction Pipelining

Imagine a waiting line of jobs, waiting to move through a series of steps, as if on an assembly line.

The non-pipeline approach is to run the steps on one job at a time. Only one job can be active at a time, the active job must move through all the steps to completion before the next job can start. When a job is at a particular step, the other steps are idle doing nothing.

The pipelined approach is to allow the next job in the queue to start step 1 as soon as the previous job has finished and moved on to step 2. In the best case scenario, the pipeline is filled with jobs at different steps or stages, all stages are working at the same time on different jobs.

Instruction pipelining is applying the pipeline approach to the von Neuman instruction execution cycle. Each instruction is a job waiting to be processed, and each step in the cycle – fetch, decode, execute, write back – is a step or stage in the pipeline.

Non-Computer Example: Laundry

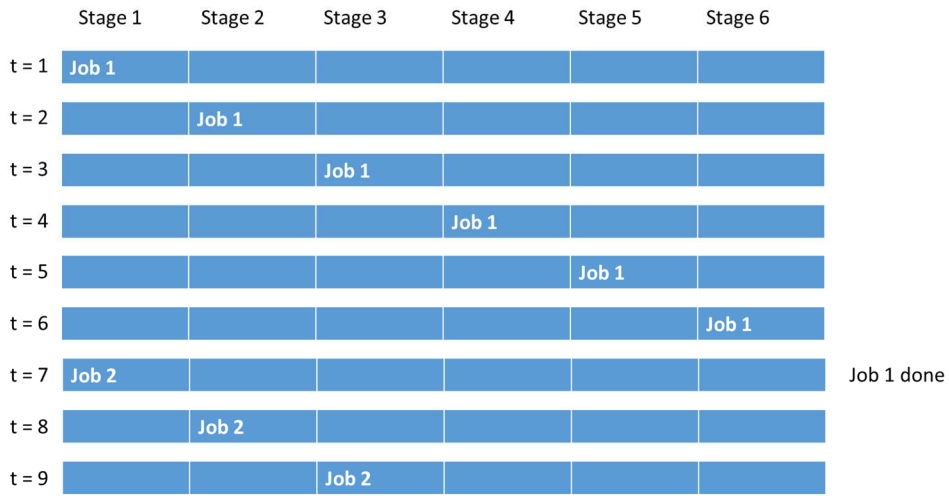
Suppose you work on a laundry assembly line where the steps are

- Presoak
- Wash
- Dry
- Iron
- Fold

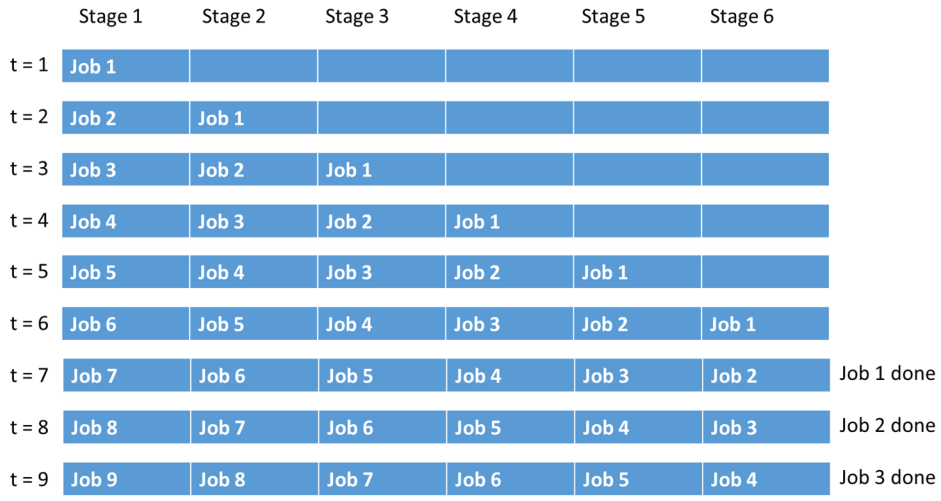
Each step takes about 30 minutes. They have to be done in this order, and there's not much opportunity for speeding the process up by doing a task in parallel for the same job (you can't start the drying stage until the washing stage is complete). Multiple jobs queue up at the front of the line. One job enters the processing at a time, and you can't start the next job until the current job is finished. While the current job is washing for example, the soak tub, dryer, ironing board, and folding table are all idle and unused. You can tell your customers to expect 1 job to be completed every 150 minutes.

With a little internal reorganization, and maybe an extra helper on the assembly line, you can implement a new policy to let the next job start the presoak stage as soon as the current job moves from presoak to wash, and so on. In this version, once the pipeline fills up, all the stages are busy all the time, with one job in presoak, the job in front of it in wash, the job in front of it in dry, and so on. With respect to performance, there's a big improvement, a new completed job appears at the output every 30 minutes. The line of jobs now moves down the assembly line 5 times faster than before. More generally, the rate of execution is n times faster if there are n stages in the pipeline.

Non-Pipelined Approach



Pipelined Approach



Non-Pipelined Performance

One job finishes every 6 time steps

Pipelined Performance

One job finishes every 1 time step

Assumptions

- No dependencies between jobs
- Each step takes about the same amount of time
- Order of jobs in the queue is known and fixed
- All of these assumptions can be violated in practice.
- Pipeline still works, but performance will be less than optimal.

Branches and Pipelines

Normal program execution is for the program counter PC to move sequentially down the list of instructions in a program in the order they are stored. We increment the PC after every cycle to move to the address of the next instruction.

But loops and decisions require branches to implement, causing instructions to execute in non-linear order. At the bottom of a loop, there will always be a backward unconditional loop to move the PC back to the top of the loop and run through the same instructions again. At the top of the loop there is a forward conditional branch. The loop continues when the branch fails and ends when the branch succeeds.

What happens to an instruction pipeline in the presence of branches? If there were no branches every program would start at the top, move steadily and uniformly downward, executing each instruction in order until reaching the last instruction. The pipeline works pretty much perfectly in this case.

But what about a program with branches?

Let's look at a simple example in a generic assembly language (no specific CPU). We'll start with a C program that adds the content of an array.

C source:

```
#include <stdio.h>
#include <stdlib.h>

int a[1000];                          // global variables
int y,z,i;

void main(void) {                     // main function
      z = 0;
      for (i=0;i<1000;i++) {          // for loop
            y = a[i];
            z = z + y;
      }
}
```

The Intel Core assembly language listing is given below:

```
; Microsoft (R) Optimizing Compiler Version 19.00.23026.0

_DATA SEGMENT                                   ; layout of global variables
COMM  _a:DWORD:03e8H
COMM  _y:DWORD
COMM  _z:DWORD
COMM  _i:DWORD
_DATA ENDS
PUBLIC      _main
_TEXT SEGMENT                                   ; code starts here
_main PROC

1     push  ebp                                 ; adjust stack at fn start
2     mov   ebp, esp
3     mov   DWORD PTR _z, 0                     ; set z to 0
4     mov   DWORD PTR _i, 0                     ; set i to 0
5     jmp   SHORT $LN4@main                     ; jump to top of loop

$LN2@main:                                      ; loop increment block
6     mov   eax, DWORD PTR _i
7     add   eax, 1
8     mov   DWORD PTR _i, eax                   ; i++

$LN4@main:                                      ; top of loop
9     cmp   DWORD PTR _i, 1000                  ; cmp i to 1000
10    jge   SHORT $LN1@main                     ; cond forward jump
11    mov   ecx, DWORD PTR _i                   ; body of loop
12    mov   edx, DWORD PTR _a[ecx*4]
13    mov   DWORD PTR _y, edx                   ; y = a[i]
14    mov   eax, DWORD PTR _z
15    add   eax, DWORD PTR _y
16    mov   DWORD PTR _z, eax                   ; z = z + y
17    jmp   SHORT $LN2@main                     ; uncond backward jump
                                                ; back to loop increment

$LN1@main:                                      ; continuation after loop
18    xor   eax, eax
19    pop   ebp                                 ; adjust stack at fn end
20    ret   0

_main ENDP
_TEXT ENDS
END
```

See http://www.eecg.toronto.edu/~amza/www.mindsec.com/files/x86regs.html for a quick guide to Intel x86 assembly language. From this guide:

- Rather than separate load and store instructions, there is a bi-directional “mov” instruction
- 32 bit General purpose registers: eax, ebx, ecx, edx
- Stack Base and Stack Pointer: ebp, esp; stack is adjusted at the beginning and end of every function call
- Data types: DWORD PTR, SHORT
- Autogen labels: $LN1@main, $LN2@main, …

Optimal Pipeline Performance

When the CPU is in the middle of executing a long series of instructions with no braches, the instruction pipeline fills and execution performance reaches its maximum. One instruction is completed every pipeline cycle. Note that we are ignoring the problem of pipeline hazards which can decrease performance for now.

Branches, both unconditional and conditional, will degrade the performance of an instruction pipeline unless special corrective measures are taken. Compilers for pipelined processors may try to minimize the number of branches to try to reduce the penalty.

From the assembled program above, zoom in on the code starting around line 3:

```
1      push   ebp                              ; adjust stack at fn start
2      mov    ebp, esp
3      mov    DWORD PTR _z, 0                  ; set z to 0
4      mov    DWORD PTR _i, 0                  ; set i to 0
5      jmp    SHORT $LN4@main                  ; jump to top of loop

$LN2@main:                                     ; loop increment block
6      mov    eax, DWORD PTR _i
7      add    eax, 1
8      mov    DWORD PTR _i, eax                ; i++

$LN4@main:                                     ; top of loop
9      cmp    DWORD PTR _i, 1000               ; cmp i to 1000
```

Let's look at a simple version of the pipeline when instruction 3 enters the pipeline:

Time	Fetch	Decode	Execute	Write Back
1	1 push			
2	2 mov	1 push		
3	3 mov	2 mov	1 push	

Now let's let the execution move forward several cycles:

Time	Fetch	Decode	Execute	Write Back
1	1 push			
2	2 mov	1 push		
3	3 mov	2 mov	1 push	
4	4 mov	3 mov	2 mov	1 push
5	5 jmp	4 mov	3 mov	2 mov
6	6 mov	5 jmp	4 mov	3 mov
7				

At step 5, instruction 5 jmp has entered the pipeline. It is an unconditional jump, but this fact won't be detected during the fetch step, which merely moves the next instruction from RAM (or cache) to the CPU. Only when instruction 5 has reached the decode stage can the type of the instruction be recognized, and the target of the jump might not be available until the execute step. But by the time instruction 5 reaches decode, the next instruction in sequence – instruction 6 – has already entered the pipeline. But instruction 6 is **not** the next instruction, **instruction 9** is.

So there may be a hiccup in the operation of the pipeline to correct for the incorrect insertion of instruction 6 into the pipeline.

Time	Fetch	Decode	Execute	Write Back	
1	1 push				
2	2 mov	1 push			
3	3 mov	2 mov	1 push		
4	4 mov	3 mov	2 mov	1 push	
5	5 jmp	4 mov	3 mov	2 mov	
6	6 mov	**5 jmp**	4 mov	3 mov	← 5 jmp detected here
7	9 cmp	~~6 mov~~	5 jmp	4 mov	← 6 mov invalid, 9 cmp fetched

What is displayed here is a very **generic possible response**, the detailed response of a specific CPU pipeline can vary. Some pipelines may have added hardware to avoid the kind of one-cycle time penalty shown at time 7.

A similar problem with optimal pipeline operation occurs with a conditional branch but in a different way. A conditional branch may or may not branch, depending on the condition codes set by the previous instruction. Similarly to the unconditional branch, the CPU can access the op code and determine that the instruction is a branch during the decode step. But unlike an unconditional branch, there are two possible next instructions: the contiguous instruction in the next memory location (if the branch does not succeed), or a different instruction at a different address stored in the branch opcode (if the branch succeeds). Also unlike the unconditional branch, the result of the branch (whether it will be taken or not) cannot be determined until the execute step.

Look at the program starting at instruction 10:

```
9     cmp   DWORD PTR _i, 1000                ; cmp i to 1000
10    jge   SHORT $LN1@main                   ; cond forward jump
11    mov   ecx, DWORD PTR _i                 ; body of loop
12    mov   edx, DWORD PTR _a[ecx*4]
13    mov   DWORD PTR _y, edx                 ; y = a[i]
14    mov   eax, DWORD PTR _z
15    add   eax, DWORD PTR _y
16    mov   DWORD PTR _z, eax                 ; z = z + y
17    jmp   SHORT $LN2@main                   ; uncond backward jump
                                              ; back to loop increment

$LN1@main:                                    ; continuation after loop
18    xor   eax, eax
19    pop   ebp                               ; adjust stack at fn end
20    ret   0
```

Branch fails, jump **is not** taken, instruction 11 follows instruction 10:

Time	Fetch	Decode	Execute	Write Back	
X	11 mov	10 jge	9 cmp	…	
X + 1	12 mov	11 mov	**10 jge**	9 cmp	**← result of jge computed**
X + 2	13 mov	12 mov	**11 mov**	10 jge	**← no special action required**

Branch succeeds, jump **is** taken, instruction 18 follows instruction 10:

Time	Fetch	Decode	Execute	Write Back	
X	11 mov	10 jge	9 cmp	…	
X + 1	12 mov	11 mov	**10 jge**	9 cmp	**← result of jge computed**
X + 2	**18 xor**	~~12 mov~~	~~11 mov~~	10 jge	**← flush pipeline to invalidate incorrectly fetched instructions and start over with 18 xor**

Ways to Fix

- Branch Prediction
- Multiple Pipelines

Branch Prediction

When a conditional branch instruction is decoded, there are two possible "next" instructions: the instruction that immediately follows in memory if the branch is not taken, and the target or operand of the branch if the branch is taken. If the branch is part of an if-else, there may be no way to predict which branch will be taken. But if the branch is part of loop, there may be. A conditional forward branch at the top of a loop will fail more often than it will succeed. As long as the loop is continuing the branch will fail (e.g., the loop counter has not reached its maximum value). When the loop counter reaches its limit the branch succeeds. It is this branch that terminates the loop and moves execution on to the instructions following the loop. Most loops run for 10, 100, or even 1000 iterations. So for a loop that executes 1000 times, the branch will fail 999 times and will succeed 1 time. It would make sense in a case like this to always guess that the branch will fail. You will be right 999 times out of 1000 and wrong only 1 time in a 1000 (or however many times the loop executes).

Branch prediction requires a small number of bits added to the instruction pipeline to track the state of a conditional branch in the pipeline. Prediction is implemented by using a FSM (finite state machine) to remember a small amount of history about that instruction the last time it was executed. If the branch failed the last time it was checked, the pipeline can guess that it will do the same thing the next time it is encountered.

Exactly where to store these bits is not standardized. It could be part of the instruction cache, or it could be a specialized lookup table as part of the pipeline management hardware. If bits are stored in cache, history is lost if the instruction gets kicked out of cache. The recommended implementation is a **branch history table** that is maintained by the instruction pipeline.

The branch history table is a cache used by the pipeline to make decisions about branches as they are encountered. It holds three fields for each branch

- Address
- History bits
- Target address (where to branch to if branch succeeds)

When a branch is encountered for the first time, it is not in the table, so it must be entered and the prediction bits initialized. Based on the initial state, a static guess is made about the branch, and the target address is left blank since the instruction has not been decoded yet. Once the branch actually executes, the result of the branch is updated in the history bits, and the target address is filled in. If the instruction is encountered a second time, a better guess about its execution can be made, and its target address is now available in the table.

Types of Prediction

- Static Guess (No History Bits): Always guess that a branch is either taken or not taken.
- One Bit History: If the bit is cleared, guess branch not taken. If the bit is set, guess taken. Update the bit based on previous execution.
- Two Bit History: This requires two incorrect guesses before changing the guess for the next branch.

Multiple Pipelines

Another approach is to use multiple pipelines so that a single pipeline with a conditional branch can be “forked” to become two pipelines, one with the next instruction based on a branch fail and the other pipeline based on a branch success. The extra pipeline sits unused until a conditional branch is encountered. Then the two versions of the pipeline are run in parallel even though only one will prove to be valid. As soon as the result of the branch is known, the correct pipeline continues execution, and the duplicate pipeline is invalidated and returns to its backup state. If the forked pipeline encounters another conditional branch, some systems will fork both of these pipelines to create 4 pipelines until it is known which pipeline contains the valid instructions.