## A Trip to the Laundry Room

To illustrate the principle of pipelining, we will take a look at the dreaded chore: laundry.

Buddy, Holly, Michael and Jackson live in the same apartment building and because of work schedules and busy social calendars must do their laundry on Tuesday evenings. The apartment has one washer, one dryer and one folding station. The washer takes 30 minutes, the dryer takes another 30 minutes and it takes 15 minutes to fold the laundry and 15 minutes to put the laundry in a basket and take it back upstairs. If each has one load of laundry to wash, how long will it take...

1. If they do laundry sequentially (one person in the laundry room at a time).

| **Person** | **6:30p** | **6:45p** | **7:00p** | **7:15p** | **7:30p** | **7:45p** | **8:00p** | **8:15p** | **8:30p** | **8:45p** | **9:00p** | **9:15p** | **9:30p** | **9:45p** | **10:00p** | **10:15p** | **10:30p** | **10:45p** | **11:00p** | **11:15p** | **11:30p** | **11:45p** | **12:00a** | **12:15a** | **12:30a** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Buddy | WASH | | DRY | | FOLD | STASH | | | | | | | | | | | | | | | | | | | |
| Holly | | | | | | | WASH | | DRY | | FOLD | STASH | | | | | | | | | | | | | |
| Michael | | | | | | | | | | | | | WASH | | DRY | | FOLD | STASH | | | | | | | |
| Jackson | | | | | | | | | | | | | | | | | | | WASH | | DRY | | FOLD | STASH | |

**6 HOURS!!!**

2. If the start their clothes washing as soon as the washer becomes available.

| **Person** | **6:30p** | **6:45p** | **7:00p** | **7:15p** | **7:30p** | **7:45p** | **8:00p** | **8:15p** | **8:30p** | **8:45p** | **9:00p** | **9:15p** | **9:30p** | **9:45p** | **10:00p** | **10:15p** | **10:30p** | **10:45p** | **11:00p** | **11:15p** | **11:30p** | **11:45p** | **12:00a** | **12:15a** | **12:30a** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Buddy | WASH | | DRY | | FOLD | STASH | | | | | | | | | | | | | | | | | | | |
| Holly | | | WASH | | DRY | | FOLD | STASH | | | | | | | | | | | | | | | | | |
| Michael | | | | | WASH | | DRY | | FOLD | STASH | | | | | | | | | | | | | | | |
| Jackson | | | | | | | WASH | | DRY | | FOLD | STASH | | | | | | | | | | | | | |

**Only 3 Hours!**

3. If the heating element is broken on the dryer and it takes twice as long to dry clothes

| **Person** | **6:30p** | **6:45p** | **7:00p** | **7:15p** | **7:30p** | **7:45p** | **8:00p** | **8:15p** | **8:30p** | **8:45p** | **9:00p** | **9:15p** | **9:30p** | **9:45p** | **10:00p** | **10:15p** | **10:30p** | **10:45p** | **11:00p** | **11:15p** | **11:30p** | **11:45p** | **12:00a** | **12:15a** | **12:30a** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Buddy | WASH | | DRY | | | | FOLD | STASH | | | | | | | | | | | | | | | | | |
| Holly | | | WASH | | | | DRY | | | | FOLD | STASH | | | | | | | | | | | | | |
| Michael | | | | | | | WASH | | | | DRY | | | | FOLD | STASH | | | | | | | | | |
| Jackson | | | | | | | | | | | WASH | | | | DRY | | | | FOLD | STASH | | | | | |

**5 Hours!**

4. If the next person does the unthinkable and takes laundry out of the washer and PUTS IT ON TOP OF THE DRYER

| **Person** | **6:30p** | **6:45p** | **7:00p** | **7:15p** | **7:30p** | **7:45p** | **8:00p** | **8:15p** | **8:30p** | **8:45p** | **9:00p** | **9:15p** | **9:30p** | **9:45p** | **10:00p** | **10:15p** | **10:30p** | **10:45p** | **11:00p** | **11:15p** | **11:30p** | **11:45p** | **12:00a** | **12:15a** | **12:30a** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Buddy | WASH | | DRY | | | | FOLD | STASH | | | | | | | | | | | | | | | | | |
| Holly | | | WASH | | STACK | | DRY | | | | FOLD | STASH | | | | | | | | | | | | | |
| Michael | | | | | WASH | | STACK | | | | DRY | | | | FOLD | STASH | | | | | | | | | |
| Jackson | | | | | | | WASH | | | | STACK | | | | DRY | | | | FOLD | STASH | | | | | |

**Still takes five hours, but the washer is available sooner**

## Latency vs. Throughput

**latency** is the time it takes to accomplish one task. In the case of our laundry room, it is the time it takes to wash,dry,fold and stash laundry.

**throughput** is the number of tasks completed per unit of time.

The latency + throughput in our examples above:

1. latency: 1.5 hours average throughput: (4 / 6) = .67 loads/hour
2. latency: 1.5 hours average throughput: (4 / 3) = 1.33 loads/hour
3. latency: 2 hours average throughput: (4 / 5) = .8 loads/hour
4. latency: 2 hours average throughput: (4 / 5) = .8 loads/hour

Example: Consider an unpipelined system that takes 320ps (10^-12s) to complete an instruction. Compute the throughput:

```
Throughput = (1 instruction/320ps) * (1000ps/1ns)
              = .003125 i/ps * 1000ps/ns
                  = 3.12 i/ns
           =3.12 billion instructions/second=GIPS
```

In this examples, the *clock* would cycle every 320ps.

Key observations

- Pipelining does not change latency, but can improve throughput
- The potential for speedup is limited by the number of unique tasks. We call each unique task a ***pipeline stage***
- Overall improvement is limited by the slowest pipeline stage
- Unbalanced stage lengths, reduces speeded
- When there is resouce competition, we must stall in order to maintain the pipeline

## Computational Pipelining

Lets say that our throughput example from above comes from a system that has three unique computational tasks that each take 100ps to complete and then takes 20ps to update the register file. It might look something like this:

```
                  300 ps              20ps
[    COMBINATIONAL LOGIC     ]  -> [  reg ]
[  A  ]-> [ B   ] -> [   C   ]
 100ps      100ps       100ps
```

By introducing *pipeline registers* for maintaining state between each unique computational task, we can build a pipelined system with stages A, B and C. We call this a *3-stage pipeline*:

```
       100ps        20ps            100ps            20ps            100ps            20
 [ COMB. LOGIC A] -> [ P.Reg1 ] -> [COMB. LOGIC B]-> [P.Reg2] - > [COMB. LOGIC C]->[P.Reg 3]
```

Question:How often does the clocl cycle? What is the latency and throughput for this system?

```
The clock cycles in a wave pattern from high to low:
   |     |      |
   ------       ------
```

```
Timing must be synchronized so that registers are fully updated and ready to become input for the next instruction when the clock cycles from low-to-high.  Therefore, the clock must cycle every 120ps.


Latency is the time for one instruction:

   (100)+(20)+(100)+(20)+(100)+(20) = 360ps

Throughput:

        1 op                         1000ns
        -----------------      *  --------------   = 8.33 GOPS
            120ps                     1ps
```

When each stage of the pipeline has the same latency, we call it a *uniform* pipeline or *uniform partitioned* pipeline. However, this is rarely the case in modern computer systems. It is still possible to build a pipelined system when different stages have different latency, but the overall performance of the pipeline will be bounded by the slowest stage.

Example We have a three stage pipeline. Stage A takes 50ps, Stage B 150ps and Stage C 100ps. Register loading takes 20ps. What is the clock cycle, latency and throughput of such a system?

```
Our clock must cycle based on the longest delay which occurs in Stage B.

  Clock = (150ps) + (20ps) = 170ps

latency will increase because each stage must be synchronized with the clock which is timed off of the slowest
   pipeline stage:

 Latency = nClock  where n is the number of stages:
               (3)(170ps) =  510ps

 Throughput =          1 operation               1000ps
                      -----------------   *     -------------   =  5.88 GOPS
                          170ps                     1ns
```

Example: Draw a pipeline diagram for this system as it executes 3 instructions.

| **Instruction** | **CC 1** | **CC 2** | **CC 3** | **CC 4** | **CC 5** |
|---|---|---|---|---|---|
| I1 | A | B | C | | |
| I2 | | A | B | C | |
| I3 | | | A | B | C |

Example: Suppose we have 6 unique computational units that execute sequentially, named A-F and having delays of 80ps, 30ps, 60ps, 50ps, 70ps and 10ps, respectively. Register delay in this system is 20ps.

1. Where should we insert a register to create a two-stage pipeline? What would be the throughput and the latency?

```
       if we place the register between C and D, the delays are:

        stage 1: A+B+C+r = 80+30+60+20 = 190ps
        stage 2: D+E+F+r = 50+70+10+20 = 150ps

       latency = n*Clock
         our clock will have to cycle at 190ps, so latency = 2*(190) = 380ps

       Throughput =      1 op                   1000ps
                     -----------      *       ----------- = 5.26 GOPS
                           190ps                  1ns
```

2. Where should we insert two registers to create a three-stage pipeline? What would be the throughput and the latency?

```
     if we place our pipeline registers between B&C and between D&E, the delays are:

        stage 1: A+B+r = 80+30+20 = 130ps
        stage 2: C+D+r = 60+50+20 = 130ps
        stage 3: E+F = 70+10+20 = 100ps

        latency = n*Clock
          clock will cycle at 130ps, so latency = 3*130 = 390ps

       Throughput =   1 op             1000ps
                    ---------  *     ----------    = 7.69 GOPS
                      130ps              1ns
```

3. What is the throughput and latency of a 5-stage pipeline implementation?

```
     stage 1: A+r = 100ps
     stage 2: B+r =  50ps
     stage 3: C+r = 80ps
     stage 4: D+r = 70ps
     stage 5: E+F+r =100ps

     latency = n*Clock
       clock will cycle at 100ps, so latency = 5*100ps = 500ps

     Throughput =   1 op        1000ps
                 ---------  * ------------   =  10.00 GOPS
                   100ps           ns
```

Making our y86 instruction cycle fully pipelined gives us a 6-stage pipeline:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| FETCH | DECODE | EXECUTE | MEMORY | WRITE BACK | PC UPDATE | | | | |
| | FETCH | DECODE | EXECUTE | MEMORY | WRITE BACK | PC UPDATE | | | |
| | | FETCH | DECODE | EXECUTE | MEMORY | WRITE BACK | PC UPDATE | | |
| | | | FETCH | DECODE | EXECUTE | MEMORY | WRITE BACK | PC UPDATE | |
| | | | | FETCH | DECODE | EXECUTE | MEMORY | WRITE BACK | PC UPDATE |

Question: What is the obvious problem here?
The Program Counter is not updated until stage 6!, so how can we fetch the next instruction?

Pipelined systems make use of *prediction* to guess what the next instruction address will be. Remember in Y86, valP is computed during the fetch stage. Depending on the instruction, we might make use of valP to *predict* the next instruction. Often we will be correct. When are we not going to be correct?

- return statements
- jumps/branches
- procedure calls

any instruction that changes or ignores valP in a later stage of the instruction cycle has the potential to gum up this approach. So depending on the instruction, there may be a different *branch prediction* strategy designed into the pipelined system.

Clearly, we can make an incorrect prediction, but this is not the only place that our pipelined system can encounter problems. We describe potential issues in our pipeline through *dependencies* and *hazards*

## Dependencies and Hazards

Consider the following y86 code. The table that follows illustrates what happens in a 5-stage pipeline of our instruction cycle with Write back and PC Update taking palce in the final stage

```
        irmovl $50, %eax
        addl %eax, %ebx
        mrmovl 100(%ebx), %edx
```

| **Instruction** | **t1** | **t2** | **t3** | **t4** | **t5** | **t6** | **t7** | **t8** |
|---|---|---|---|---|---|---|---|---|
| irmovl $50, %eax | FETCH: icode:irmovl rB=%eax valC=50 valP=PC+6 | DECODE | EXECUTE: valE=0+valC | MEMORY | WRITEBACK %eax=50 PC=valP | | | |
| addl %eax, %ebx | | FETCH: icode:op,ifun:addl rA=%eax,rB=%ebx valP=PC+2 | DECODE valA=???? valB=%ebx | EXECUTE: valE=valA+valB | MEMORY | WRITEBACK %ebx=valE PC=valP | | |
| mrmovl 100(%ebx), %edx | | | FETCH: icode:mrmovl rA=edx,rB=%ebx valC=100 valP=PC+6 | DECODE valB=???? | EXECUTE: valE=valC+valB | MEMORY valM=M[valE] | WRITEBACK %edx=valM PC=valP | |

There is a **data dependecy** between instructions one and two (%eax) and between instructions two and three (%ebx). Instruction two cannot execute in the following clock cycle, nor can instructoin three. When an instruction cannot be executed in the next clock cycle because there is a dependecy, it is called a **hazard**

There are *three* types of hazards/dependencies:

1. **Structural Hazards**
2. **Data Hazards**
3. **Control Hazards**

**Structural Hazards**
*Structural Hazards* occur when two or more intructions need the same hardware unit at the same time.

Example: Consider a pipelined system with that shares a single port for accessing memory. The following table demonstrates the structural hazard that occurs when executing the following instructions:

```
        mrmovl          24(%ebx), %eax
        addl            $esi, %edx
        addl            %ebp, %ecx
        addl            %edx, %esi
```

| **Instruction** | **t1** | **t2** | **t3** | **t4** | **t5** | **t6** | **t7** | **t8** |
|---|---|---|---|---|---|---|---|---|
| mrmovl 24(%ebx), %eax | FETCH: icode:mrmovl rA=%eax,rB=%ebx valC=24 valP=PC+6 | DECODE: valB=R[%ebx] | EXECUTE: valE=24+valB | MEMORY: valM=M[valE] | WRITEBACK: %eax=valM PC=valP | | | |
| addl %esi, %edx | | FETCH: icode:op,ifun:addl rA=%esi,rB=%edx valP=PC+2 | DECODE: valA=R[%esi] valB=R[%edx] | EXECUTE: valE=valA+valB | MEMORY: | WRITEBACK: %edx=valE PC=valP | | |
| addl %ebp,%ecx | | | FETCH: icode:op,ifun:addl rA=%ebp,rB=%ecx valP=PC+2 | DECODE: valA=R[%ebp] valB=R[%ecx] | EXECUTE: valE=valA+valB | MEMORY: | WRITEBACK: %ecx=valE PC=valP | |
| addl %eax,%esi | | | | FETCH: icode:op,ifun:addl rA=%eax,rB=%esi valP=PC+2 | DECODE: valA=R[%eax] valB=R[%esi] | EXECUTE: valE=valA+valB | MEMORY: | WRITEBACK: %esi=valE PC=valP |

Instructions #1 and #4 require access to the same hardware unit at the same time. This is a ***structural dependecy*** that causes a *structural hazard* in the pipeline.

**Possible Solutions?**

- Different hardware: dual-port memory access.
- Stalling ( "bubble" the pipeline)

| **Instruction** | **t1** | **t2** | **t3** | **t4** | **t5** | **t6** | **t7** | **t8** | **t9** |
|---|---|---|---|---|---|---|---|---|---|
| mrmovl 24(%ebx), %eax | FETCH: icode:mrmovl rA=%eax,rB=%ebx valC=24 valP=PC+6 | DECODE: valB=R[%ebx] | EXECUTE: valE=24+valB | MEMORY: valM=M[valE] | WRITEBACK: %eax=valM PC=valP | | | | |
| addl %esi, %edx | | FETCH: icode:op,ifun:addl rA=%esi,rB=%edx valP=PC+2 | DECODE: valA=R[%esi] valB=R[%edx] | EXECUTE: valE=valA+valB | MEMORY: | WRITEBACK: %edx=valE PC=valP | | | |
| addl %ebp,%ecx | | | FETCH: icode:op,ifun:addl rA=%ebp,rB=%ecx valP=PC+2 | DECODE: valA=R[%ebp] valB=R[%ecx] | EXECUTE: valE=valA+valB | MEMORY: | WRITEBACK: %ecx=valE PC=valP | | |
| addl %eax,%esi | | | | ~~STALL~~ | FETCH: icode:op,ifun:addl rA=%eax,rB=%esi valP=PC+2 | DECODE: valA=R[%eax] valB=R[%esi] | EXECUTE: valE=valA+valB | MEMORY: | WRITEBACK: %esi=valE PC=valP |

**Data Hazards**

*Data Hazards* occur when a data dependency exists between two or more instructions. Our first example, demonstrated a data dependency that introduces a *data hazard* to the pipeline. There are *three* types of data hazards. Consider instruction *I_a* and *I_b* such that instruction *I_a* executes before instruction *I_b*:

- **Read-After-Write (RAW)**: Read-after-write data hazards occur when *I_b* attemps to read a data source *before I_a* writes its updated value. *I_b* will be incorrectly set to the old value.

  In our first example, the data dependecy between instructions 1&2 and between instructoins 2 & 3 would created a RAW Data Hazard. Instruction 2 attempts to read the value of %eax during the decode stage before the writeback stage has completed for instruction #1. The same thing happens between instructions 2&3 with register %ebx.

- **Write-After-Write (WAW)**: A write-after-write data hazard occurs when *I_b* attempts to write an operand before *I_a* has written the operand. This results in the write operations occuring out-of-order leaving the value written by *I_a* instead of the value written by *I_b*. WAW data hazards cannot occur in y86 because we only allow memory writes to occur in one stage. If more than one stage allows writes, then the potential for WAW hazards exist. It could be introduced if our pipeline supports *concurrent execution* with shared memory and register files, the following might cause a WAW data hhazard:

```
        addl        %eax, %ecx
        addl        %ebx, %ecx
```

- **Write-After-Read (WAR)**: A write-after-read data hazard occurs when *I_b* attempts to write to a destination before it is read by *I_a* causing *I_a* to use the updated value when it should use the previous value. If we modify our y86 system to support *concurrent execution* with shared memory and register files, the following might cause a WAR data hazard:

```
    addl    %eax, %ecx
    addl    %ebx, %eax
```

**Possible Solutions?**

- Stallng
- Reorder the instructions
- Data forwarding: Add hardware support/control logic for directly forwarding output from one stage to replace values from another stage.

Example:

```
irmovl        $10, %edx
irmovl        $3, %eax
addl          %edx, %eax
```

| Instruction | t1 | t2 | t3 | t4 | t5 | t6 | t7 |
|---|---|---|---|---|---|---|---|
| irmovl $10, %edx | FETCH | DECODE | EXECUTE | MEMORY | WRITEBACK | | |
| irmovl $3, %eax | | FETCH | DECODE | EXECUTE | MEMORY | WRITEBACK | |
| addl %edx, %eax | | | FETCH | DECODE | EXECUTE | MEMORY | WRITEBACK |

Instruction 3 has a data dependecy on instruction 1 (%edx) and on instruction 2 (%eax) but neither instruction has completed the writeback stage. However, if we forward the values from the memory pipeline register and from the execute pipeline register to the decode pipeline register, then the instruction will complete as normal.

The key thing to remember about data forwarding is that you cannot forward data back in time.

Example:

```
mrmovl  $8(%ebp), %eax
addl    %eax, %ecx
andl    %eax, %edx
```

| Instruction | t1 | t2 | t3 | t4 | t5 | t6 | t7 |
|---|---|---|---|---|---|---|---|
| irmovl $8(%ebp), %eax | FETCH | DECODE | EXECUTE | MEMORY | WRITEBACK | | |
| addl %eax, %ecx | | FETCH | DECODE | EXECUTE | MEMORY | WRITEBACK | |
| andl %eax, %edx | | | FETCH | DECODE | EXECUTE | MEMORY | WRITEBACK |

We cannot forward the value for %eax from instruction 1 because it is not loaded from memory until the Memory stage completes, however instruction 2 has already entered the Execute stage by then. In this case, we have to stall.

| Instruction | t1 | t2 | t3 | t4 | t5 | t6 | t7 | |
|---|---|---|---|---|---|---|---|---|
| irmovl $8(%ebp), %eax | FETCH | DECODE | EXECUTE | MEMORY | WRITEBACK | | | |
| addl %eax, %ecx | | STALL | FETCH | DECODE | EXECUTE | MEMORY | WRITEBACK | |
| andl %eax, %edx | | | | FETCH | DECODE | EXECUTE | MEMORY | WRITEBACK |

**Control Hazards**

*Control Hazards* can occur when the outcome of some conditional test determines whether or not the next instruction will execute.

Example:

```
subl    %eax, %ecx
je      elsewhere
irmovl  $1, %edx
```

| Instruction | t1 | t2 | t3 | t4 | t5 | t6 | t7 |
|---|---|---|---|---|---|---|---|
| subl %eax, %ecx | FETCH | DECODE | EXECUTE | MEMORY | WRITEBACK | | |
| je elswhere | | FETCH | DECODE | EXECUTE | MEMORY | WRITEBACK | |
| irmovl $1, %edx | | | FETCH | DECODE | EXECUTE | MEMORY | WRITEBACK |

The fetch for instruction 3 may create a control hazard if the jump is taken. When the jump is taken, the PC is updated based on valC and the condition registers that are set during the execute stage. Stalling is the easiest approach, but can quickly incur a penalty for pipeline performance.

| Instruction | t1 | t2 | t3 | t4 | t5 | t6 | t7 | | |
|---|---|---|---|---|---|---|---|---|---|
| subl %eax, %ecx | FETCH | DECODE | EXECUTE | MEMORY | WRITEBACK | | | | |
| je elswhere | | FETCH | DECODE | EXECUTE | MEMORY | WRITEBACK | | | |
| nop | | STALL | STALL | STALL | STALL | STALL | | | |
| nop | | | STALL | STALL | STALL | STALL | STALL | | |
| irmovl $1, %edx | | | | | FETCH | DECODE | EXECUTE | MEMORY | WRITEBACK |

Because branches/jumps are so common, we need a technique that can help minimize the penalty of branches in code. These schemes are called **branch prediction strategy**. Our Y86 implementation follows a branch prediction strategy known as *always take*. Y86 assumes that a branch will always be followed, this is a simple implementation since we can use valC from the fetch stage as our predicted address and in-general, this strategy is correct about 60% of time.

Example: Always taken, Correct Prediction:

```
  subl        %eax, %ecx
  je          elsewhere
  irmvol      $1, $eax
  irmovl      $0, %esi
  elsewhere:
  addl        %esi, %eax
  push        %ebp
```

| Instruction | t1 | t2 | t3 | t4 | t5 | t6 | t7 |
|---|---|---|---|---|---|---|---|
| subl %eax, %ecx | FETCH | DECODE | EXECUTE | MEMORY | WRITEBACK | | |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| je elswhere | | FETCH | DECODE | EXECUTE | MEMORY | WRITEBACK | |
| addl %esi, %eax | | | FETCH | DECODE | EXECUTE | MEMORY | WRITEBACK |

Here our prediciton strategy works seamlessly since the branch is correctly followed. We incur no delay due to stalling.

Example: Always taken, Wrong Prediction:

```
    subl        %eax, %ecx
    je          elsewhere
    irmvol      $1, $eax
    irmovl      $0, %esi
    elsewhere:
    addl        %esi, %eax
    push        %ebp
```

| **Instruction** | **t1** | **t2** | **t3** | **t4** | **t5** | **t6** | **t7** | |
|---|---|---|---|---|---|---|---|---|
| subl %eax, %ecx | FETCH | DECODE | EXECUTE | MEMORY | WRITEBACK | | | |
| je elswhere | | FETCH | DECODE | EXECUTE | MEMORY | WRITEBACK | | |
| addl %esi, %eax | | | FETCH | DECODE | EXECUTE | MEMORY | WRITEBACK | |
| push %ebp | | | | FETCH | DECODE | EXECUTE | MEMORY | WRITEBACK |

Here the next *two* instructions will have already been fetched before we realize that the branch prediction was incorrect. Now we will have the *flush* the pipeline and fetch the correct instruction.

| **Instruction** | **t1** | **t2** | **t3** | **t4** | **t5** | **t6** | **t7** | | |
|---|---|---|---|---|---|---|---|---|---|
| subl %eax, %ecx | FETCH | DECODE | EXECUTE | MEMORY | WRITEBACK | | | | |
| je elswhere | | FETCH | DECODE | EXECUTE | MEMORY | WRITEBACK | | | |
| addl %esi, %eax | | | FETCH | DECODE | FLUSH | FLUSH | FLUSH | | |
| push %ebp | | | | FETCH | FLUSH | FLUSH | FLUSH | FLUSH | |
| irmovl $0, %eax | | | | | FETCH | DECODE | EXECUTE | MEMORY | WRITEBACK |

Common Branch Prediction Strategies

- *Stall the Pipeline*: Simple, cheap implementation. All branch instructions (return included!) will slow the pipeline.
- *Always-Take (Predict Taken)*: Not too complicated, in general has better than 50% success rate.
- *Never-Take (Predict Not-Taken)*: ~40% success rate. Not too complicated.
- *Delayed-Branch*: Attempts to minimize the time wasted by mispredicted branches by filling the space known as the *branch delay slot(s)* with instructions that are execute whether or not the branch is taken. Good success rate, requires intervention on the part of either the programmer or the compiler to select instructions to fill the branch delay slots.

**Other options?**
Replace *conditional control* of program execution with *conditional data movement*. By assigning values conditionally, we can avoid incurring any branch penalities:

```
  int absdiff(int x, int y) {
    if (x<y)
      return y-x;
    else
      return x-y;
  }


  vs.

  int absdiff(int x, int y) {

    int tval = y-x;
    int rval = x-y;
    int test = x < y;
    if (test) rval = tval;  //  cmovl in ASM!!
    return rval;
  }
```

## Pipeline Performance

We guage our pipeline performance with CPI(Cycles per Instruction). 1.0 is the limiting value which is the ideal case where one instruction begins each clock cycle.

```
    CPI = 1.0 + ( ∑ (Freq_i * Freq_p * Penalty ) )
```

Example: After analyzing our code, we determine that load instructions account for 25% of all instructions, 20% of them cause hazards that result in stalling for 1 clock cycle. Condtional branches account for 20% of all instructions and follow an always taken strategy with 60% success rate, incuring a 2 clock cycle penalty on failure. Return statements account for 2% of the instructions and require a 3 clock cycle stall in order to flush the pipeline and complete the return call. Compute the CPI:

```
   CPI = 1.0 + ∑ Freq_i * Freq_p * Penalty
       = 1.0 + (.25 * .20 * 1) + (.20 * .40 * 2) + (.02 * 1.00 * 3)
       = 1.0 + (.05) + (.16) + (.06)
       = 1.27
```

Example:: Under the same assumptions, suppose we benchmark a new branch prediction strategy that achoeves a success rate of 65%. What is the overall impact on the CPI assuming that no other frequencies are affected?

This means a failure rate of .35 which will change our branch penalty to:

```
  (.20 *. 35 * 2) = .14
```

This will reduce the overall CPI by .02.