Pipelining

Motivation

- Need to understand how CPU does instruction execution

Consider

- Suppose we have C code like so:

```
int c = a+b;
```

- Suppose a is at address 0x1000
- And b is at 0x2000
- And c is at 0x3000

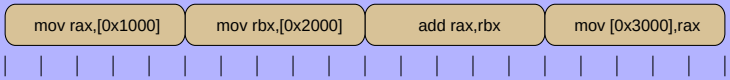
Code

- Corresponding assembly code (x64):

```
mov rax, [0x1000]
mov rbx, [0x2000]
add rax, rbx  ;rax <- rax+rbx
mov [0x3000], rax
```

Execution

- Example: Each tick mark represents 1ns

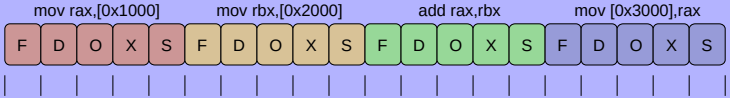


Observe

- Each instruction has several parts to its execution
  - Must get the instruction from RAM
  - Figure out what it is
  - Get the operands ready
  - Do the operation
  - Store results
- For simplicity, suppose each of these takes 1ns

Execution

- Here's a broken-down view of instruction execution
- 20ns total



Observe

- Different actions involve physically distinct silicon
- So while we are decoding instruction n, we can fetch instruction n+1
- When we are fetching instruction n+2, we can decode n+1 and fetch operands for n

Analysis

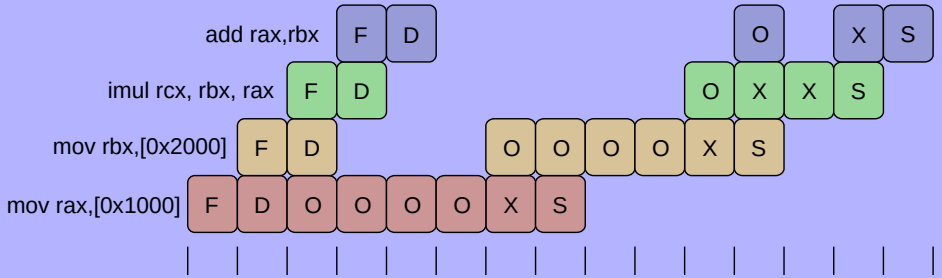
- We've performed four instructions in 8ns
- In the long run, we average one instruction *issued* every 1ns
- We've made CPU 5x faster without really changing its internals!
- This is a big win, so high performance CPU's always use this

Note

- It's unrealistic to assume every instruction takes same amount of time
- Consider this code
  mov rax, [0x1000]
  mov rbx, [0x2000]
  imul rcx,rbx,rax ;rcx <- rbx * rax
  add rax, rbx ;rax <- rax + rbx
- Suppose fetching from RAM takes four cycles, add takes 1 cycle, mul takes 2 cycles

Diagram

- Pipeline: Since one instruction is using a stage for several cycles, that blocks all subsequent instructions
- Hardware delays (*stalls*) instructions as needed

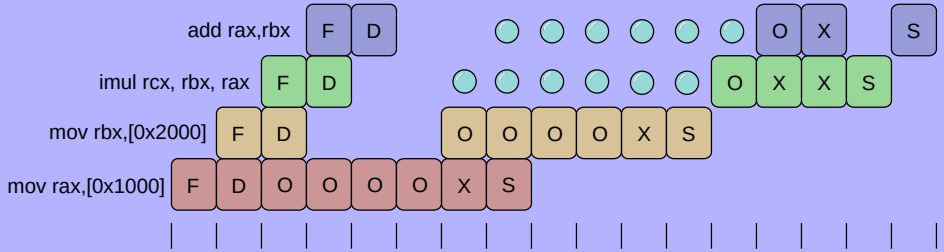


Notice

- Addition and multiplication use different hardware
- And: Memory fetches and register reads use different hardware
- So we can run instruction *out of order* and *in parallel* (superscalar)

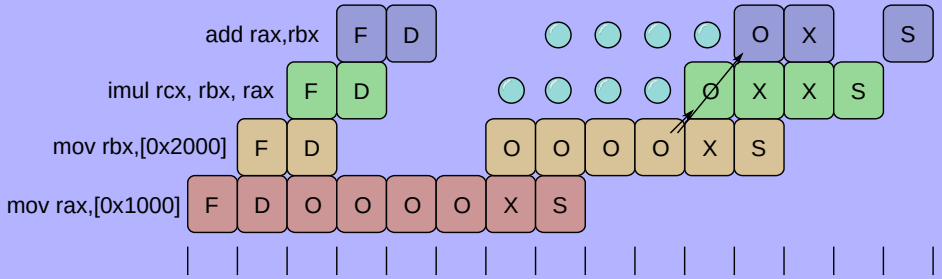
But...

- There's a problem
- The imul gets its operands from the two mov's
  - Result isn't available until the "store" pipeline stage completes
- So processor must insert a *bubble* in pipeline to delay the imul (and the add) until operands available



Forwarding

- CPU can use intrastage *forwarding* to move data from one pipeline stage directly to the stage that needs the data



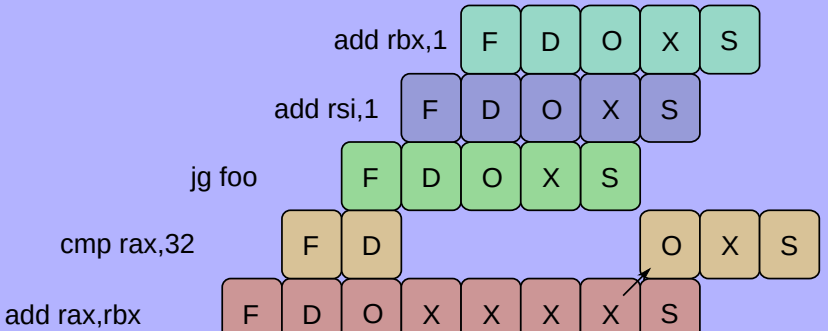
Example

- Suppose we have this code:

```
add rax,rbx
cmp rax,32
jg foo
add rax,1
```

Execution

- CPU can use *register renaming*
  - If same register used in two closely spaced instructions and no logical dependency: CPU pretends like second instruction used totally different register
- This can also result in additional out-of-order completions



Note

- Superscalar ≠ multicore
  - If multiple cores: Several instructions execute at same time
  - But superscalar = multiple instructions “in flight” *within* a single core
- Note that modern CPU’s typically have around a dozen pipeline stages and are both multicore *and* superscalar
  - They can easily handle a couple hundred out-of-order instructions

Skylake

- Real world example: Skylake can *simultaneously* do...
  - 4 add/bitwise operations
  - 2 shifts
  - 1 multiply
  - 2 bitmask
  - 1 divide

Note

- Things are more complex than just stated
- Ex: Even if CPU has only one physical multiplier, it can have two instructions at different points within the multiplier
- Distinction: Throughput vs. latency
  - Latency = time from when instruction is *fetched* to when it is *retired*
  - Throughput = If we have a long sequence of same instruction, on average, we'll get one result every n cycles

Example

- Simple operations (add, bitwise): Latency & throughput both 1
- Multiply: Latency = 3, throughput = 1
- Divide: Latency varies, depending on value; throughput might be 100 cycles!

So What?

- How does this impact our software code?
- Good compilers will typically try to arrange that instructions which use same computational units are spaced apart
- Example follows...

Example

- Suppose you write this:

```
int a = 10;
int b = 20;
int c = a*b;
int d = a * 15;
int e = a+b;
```

Example

- Compiler acts like you wrote

```
int a = 10;
int b = 20;
int c = a*b;
int e = a+b;
int d = a * 15;
```

Pro Tip

- Try to arrange code that uses the same functional unit so it's not all bunched together
- Usually, compilers will do this for us, but make sure to turn optimization on
  - And tell compiler what target architecture you want!

Branches

- Branchy code is biggest challenge for pipelined architectures.
- Consider:

```
add rax,rbx
cmp rax,32
jg foo
add rsi,1
add rbx,2
foo:
sub rcx,1
sub rdx,2
```

Operation

- Computer begins executing instructions
- But: When it comes to the jg: Doesn't yet know whether branch is taken or not
  - Not until cmp finishes its execute stage
- So it must guess which instruction to fetch next
  - (Diagram in class)

Guess

- If CPU guesses right: Keeps running at full speed
- If CPU guesses wrong: Wasted time; delay
- Also called *speculative execution*

Moral

- The moral of the story?
  - Branches hurt pipeline performance
  - Longer pipelines suffer more than shorter ones
  - CPU's use *branch prediction* to try to minimize slowdown

Branches

- Simple rule: Static prediction
- Predict: Forward branches are not taken
  - Why? Usually seen with loops
  - Consider some C code...

Code

```
for(i=0;i<10;i++){
   a=a+1;
}
```

```
mov rax,0 ;hold i in rax
loopstart:
cmp rax,10
je endloop
add rax,1
jmp loopstart
```

- The conditional jump is only taken 10% of the time
- So we usually predict not taken

Branches

- Backwards branches: Predict as taken

```
do {
   a=a+1;
   foo()
} while(a<10);
```

```
loopstart:
add rax,1
call foo
cmp rax,10
jle loopstart
```

- The branch is taken on almost every iteration

Note

- This doesn't work so well in non-loop context:

```
if( i == 1 )
   foo();
else if( i == 2 )
   bar();
else
   baz();
```

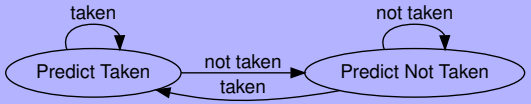
```
cmp rax,1
jne elseif1
call foo
jmp endif
elseif1:
cmp rax,2
jne else
call bar
jmp endif
else:
call baz
endif:
```

Note

- If this code is only executed once, maybe it doesn't matter much
- But what if it's inside a loop?
  - Or inside a function that's called in a loop?

Branch Prediction

- Higher performance: Use *branch prediction table*
- Idea: CPU stores table
  - One column: Instruction address (or at least low n bits of it)
  - Second column: What did we do last time we saw branch?
    - Taken or not?
- Goal: Learn from past behavior
- Essentially, we model a state machine:



Problem

- This is an improvement, but it doesn't always help
- Suppose we have code like so:

```
for(i=0;i<100;++i){
    if( i % 4 == 0 )
        foo();
    else
        bar();
}
```

Behavior

- Suppose CPU predicts “taken” (i.e., ‘else’) at first (i=0)
  - This is a misprediction
- But CPU learns and records “this branch is not taken”
- What happens when i is 1?

Behavior

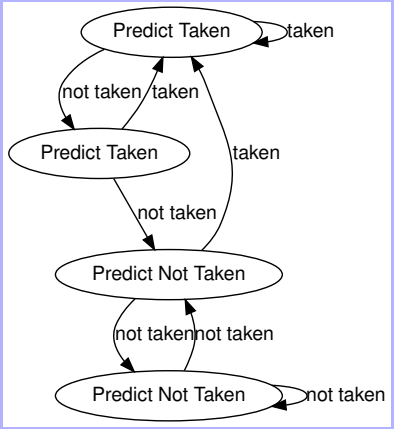
- CPU predicts “not taken”
- But that’s wrong!
  - So we have second misprediction
- CPU learns and records “this branch is taken”
- When i=2, CPU guesses correctly
- And same thing when i=3

Behavior

- But when i=4: Mispredicts
- And again when i=5 (same reason as when i=1)
- This results in only 50% correct guesses

Improvement

- CPU engineers can add states to the state machine
- This can improve prediction accuracy



Pro Tip

- Try to avoid branches especially in loops
- Might be better to write two loops!

```
for(int i=0;i<N;++i){
    if( i % 4 == 0 )
        foo(A[i]);
    else
        bar(A[i]);
}
```

```
for(int i=1;i<N;i+=4){
    bar(A[i]);
    bar(A[i+1]);
    bar(A[i+2]);
}
for(int i=0;i<N;i+=1){
    foo(A[i]);
}
```

Sources

- https://softwareengineering.stackexchange.com/questions/210818/how-long-is-a-typical-modern-microprocessor-pipel
- Intel Corp. Intel 64 and IA-32 Architectures Optimization Reference Manual.
- Michael Suess. Parallel Programming Fun with Loop Carried Dependencies. http://www.thinkingparallel.com/2007/05/02/parallel-programming-fun-with-loop-carried-depend

Created using LaTeX.

Main font: Gentium Book Basic, by Victor Gaultney. See http://software.sil.org/gentium/
Monospace font: Source Code Pro, by Paul D. Hunt. See https://fonts.google.com/specimen/Source+Code+Pro and http://sourceforge.net/adobe
Icons by Ulisse Perusin, Steven Garrity, Lapo Calamandrei, Ryan Collier, Rodney Dawes, Andreas Nilsson, Tuomas Kuosmanen, Garrett LeSage, and Jakub Steiner. See http://tango-project.org