

# **ECE 571 – Advanced Microprocessor-Based Design Lecture 4**

Vince Weaver

`http://www.eece.maine.edu/~vweaver`

`vincent.weaver@maine.edu`

28 January 2016

# Announcements

- Homework #1 was due
- Homework #2 will be posted soon. Probably a paper to read.



# Some sample code

```
int i;  
int x[128];  
  
for ( i=0; i < 128; i++ ) {  
    x[i]=0;  
}
```

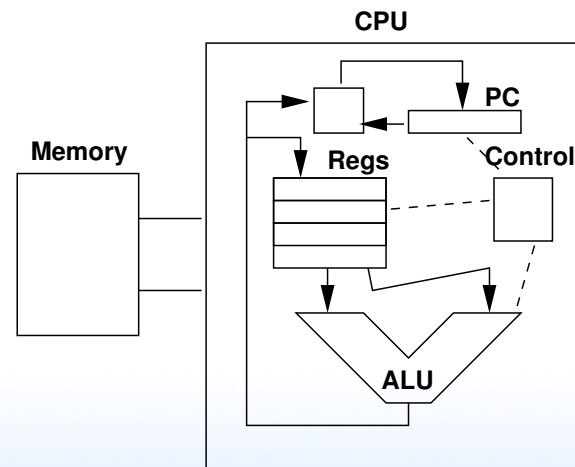


```
    mov r0,#0          ; i=0
loop:
    mov r1,x           ; point r1 to x array
    lsl r2,r0,#2       ; r2=i*4
    mov r3,0           ; want to write 0 to x[i]
    str r3,[r1,r2]     ; x[i]=0
    add r0,r0,#1       ; i++
    cmp r0,#128        ; is i==128?
    bne loop           ; if not, keep looping
```



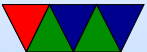
# Simple CPUs

- Ran one instruction at a time.
- Could take one or multiple cycles (IPC 1.0 or less)
- Example – single instruction take 1-5 cycles?



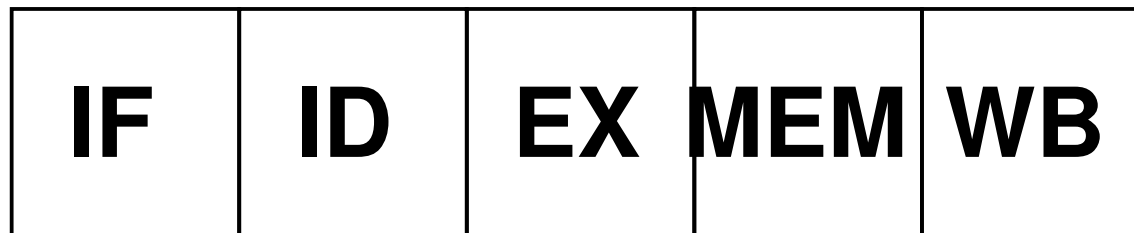
# IPC Metric

- Instructions per Cycle
- Higher is better
- Inverse of CPI (cycles per instruction)



# Pipelined CPUs

- 5-stage MIPS pipeline



# Pipelined CPUs

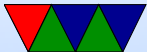
- IF = Instruction Fetch, Update PC  
Fetch 32-bit instruction from L1-cache
- ID = Decode, Fetch Register
- EX = execute (ALU, maybe shifter, multiplier, divide)  
Memory address calculated
- MEM = Memory – if memory had to be accessed, happens now.
- WB = register values written back to the register file





# Cycle 1

IF	<code>mov r0,#0</code>
ID	
EX	
MEM	
WB	



# Cycle 2

IF	<code>mov r1,x</code>
ID	<code>mov r0,#0</code>
EX	
MEM	
WB	



# Cycle 3

IF	<code>lsl r2,r0,#2</code>
ID	<code>mov r1,x</code>
EX	<code>mov r0,#0</code>
MEM	
WB	



# Cycle 4

IF	<code>mov r3,#0</code>
ID	<code>lsl r2,r0,#2</code>
EX	<code>mov r1,x</code>
MEM	<code>mov r0,#0</code>
WB	



## Cycle 5

IF	<code>str r3, [r1, r2]</code>
ID	<code>mov r3, #0</code>
EX	<code>lsl r2, r0, #2</code>
MEM	<code>mov r1, x</code>
WB	<code>mov r0, #0</code>



# Benefits/Downside

- From 2-stage to Pentium 4 31-stage
- Latency higher (5 cycles) but average might be 1 cycle
- Why bother? Can you run the clock faster?



# Data Hazards

Happen because instructions might depend on results from instructions ahead of them in the pipeline that haven't been written back yet.

- RAW – “true” dependency – problem. Bypassing?
- WAR – “anti” dependency – not a problem if commit in order
- WAW – “output” dependency – not a problem as long as ordered
- RAR – not a problem



# Structural Hazards

- CPU can't just provide. Not enough multipliers for example





# Control Hazards

- How quickly can we know outcome of a branch
- Branch prediction? Branch delay slot?



# Branch Prediction

- Predict (guess) if a branch is taken or not.
- What do we do if guess wrong? (have to have some way to cancel and start over)
- Modern predictors can be very good, greater than 99%
- Designs are complex and could fill an entire class



# Memory Delay

- Memory/cache is slow
- Need to bubble / Memory Delay Slot



# The Memory Wall

- Wulf and McKee
- Processors getting faster more quickly than memory
- Processors can spend large amounts of time waiting for memory to be available
- How do we hide this?



# Caches

- Basic idea is that you have small, faster memories that are closer to the CPU and much faster
- Data from main memory is cached in these caches
- Data is automatically brought in as needed.  
Also can be pre-fetched, either explicitly by program or by the hardware guessing.
- What are the downsides of pre-fetching?
- Modern systems often have multiple levels of cache.  
Usual a small (32k or so each) L1 instruction and data,



a larger (128k?) shared L2, then L3 and even L4.

- Modern systems also might share caches between processors, more on that later
- Again, could teach a whole class on caches



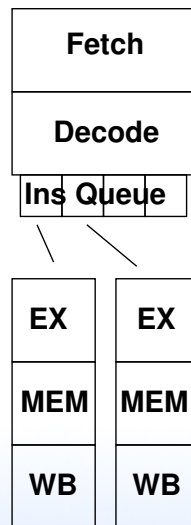
# Exploiting Parallelism

- How can we take advantage of parallelism in the control stream?
- Can we execute more than one instruction at a time?



# Multi-Issue (Super-Scalar)

- Decode up to  $X$  instructions at a time, and if no dependencies issue at same time.
- Dual issue example. Can have theoretical IPC of 2.0
- Can have unequal pipelines.





# Out-of-Order

- Tries to exploit instruction-level parallelism
- Instead of being stuck waiting for a resource to become available for an instruction (cache, multiplier, etc) keep executing instructions beyond as long as there are no dependencies
- Need to insure that instructions commit in order
- What happens on exception? (interrupt, branch mispredict, etc)



- Register Renaming
- Re-order buffer
- Speculative execution / Branch Prediction?



# Perf Counters related to Stalls

- Front-end stalls – fetch, decode, icache misses
- Back-end stalls – memory accesses



# Instruction Level Parallelism

- Using super-scalar and/or OoO (Out of Order) execution try to find parallelism within your serial code
- Chip companies want to speed up existing code. Why? (it's a pain to change, you might not have source, etc.)



# Other Ways to get better Parallelism



# SIMD / Vector Instructions

- x86: MMX/SSE/SSE2/AVX/AVX2  
semi-ralted FMA
- MMX (mostly deprecated), AMD's 3DNow!  
(deprecated)
- PowerPC AltiVec
- ARM: Neon



# SSE / x86

- SSE (streaming SIMD): 128-bit registers XMM0 - XMM7, can be used as 4 32-bit floats
- SSE2 : 2\*64bit int or float, 4 \* 32-bit int or float, 8x16 bit int, 16x8-bit int
- SSE3 : minor update, add dsp and others
- SSSE3 (the s is for supplemental): shuffle, horizontal add



- SSE4 : popcnt, dot product





# AVX / x86

- AVX (advanced vector extensions) – now 256 bits, YMM0-YMM15 low bits are the XMM registers. Now twice as many.  
Also adds three operand instructions  $a=b+c$
- AVX2 – 3 operand Fused-Multiply Add, more 256 instructions
- AVX-512 – version used on Xeon Phi (knights landing) and Skylake – now 512 bits, ZMM0-XMM31



# Multi-core

- More's law gives you lots of transistors. Hit limit of how fast to make a single processor, so why not just put more on the die?
- Exploits multi-programmed parallelism rather than instruction-level parallelism



# Multi-threaded

- SMT (simultaneous multithreading), Intel Hyperthreading
- Hybrid of multi-core and multi-pipeline
- Your pipelines might not always be full, especially if waiting on memory
- Why not duplicate fetch/decode logic, and have two programs execute at once on same set of pipelines.
- If one is idle/stalled, run instructions from other thread



- Looks to OS as if you have two cores, but really just one with two instruction dispatch stages
- Extra logic to make sure that pipelines used fairly, the results get committed to the right register file, etc.

TODO: draw diagram

