

ECE 574 – Cluster Computing

Lecture 5

Vince Weaver

`https://web.eece.maine.edu/~vweaver`

`vincent.weaver@maine.edu`

4 February 2025

Announcements

- HW#2 was posted
- If you missed getting an account slip for the homework assignment, let me know
- For now homeworks due Thursday but maybe as semester goes on move to Friday



Aside on Weather Forecasting

- Good example as even with large supercomputers models can still struggle (the recent snow forecasts)
- <https://www.noaa.gov/news-release/noaa-completes-upgrade-to-weather-and-climate-supercomputer-system>
- 2023 Upgraded two 14 PFLOP machines (peak or max?)
- Would be around 100 on top500? Why not listed?
Maybe not worth the trouble
- “American” vs European Model
- 12km grid? 1km grid? Large simulations, must finish in time to be useful



Computer Architecture Review



Parallel Computing – Single Core

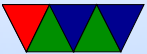
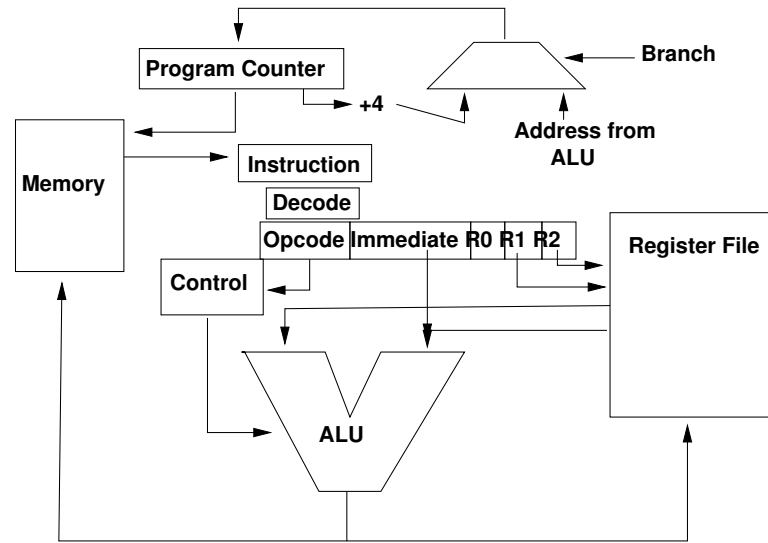
- Most code written for serial execution:
one step at a time
- You can re-write it to try to do things in parallel
(we'll get to that)
- What if the hardware could take your serial code and try
to get parallelism out of it for you?



Simple CPUs

- Small, 6502 made of 4500 transistors
- Ran one instruction at a time.
- Could take one or multiple cycles (Instructions per Cycle (IPC) 1.0 or less)
- Example – single instruction take 2-7 cycles?





Code Example – C

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



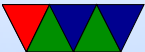
Can Compiler Provide Speedup?

- First attempt at optimization is to have compiler generate optimal assembly code
- Even today good compilers can often be beaten by skilled assembly language programmers



Code Example – ARM assembly

```
    mov r0 ,#0          ; i=0 (register allocation)
loop:
    ldr r1 ,=x          ; point r1 to X array
    lsl r2 ,r0 ,#2      ; r2=i*4
    mov r3 ,#0          ; value to store
    str r3 ,[r1 ,r2]    ; X[i]=0
    add r0 ,r0 ,#1      ; i=i+1
    cmp r0 ,#128        ; check if reached 128
    bne loop            ; loop if not equal
.bss
.lcomm x ,128 ,4        ; reserve room for 128 ints
```



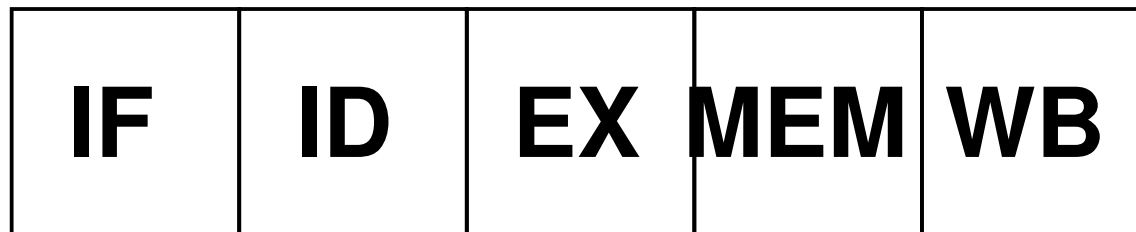
Possible Optimizations

- Constant propagation?
- Code hoisting?
- ARM optimizations
 - barrel shifter
 - fancy addressing (auto-increment)
- Loop unrolling?
- Writing 64-bits of zeros rather than 32-bits (then you have to worry about alignment)
- Your OS/libc will provide hand-optimized `memset()`



Pipelined CPUs

- 5-stage MIPS pipeline
- From 2-stage to Pentium 4 31-stage
- Example – single instruction always take 5 cycles? But what about on average? (Theoretical max IPC 1.0)



Pipelined CPUs

- IF = Instruction Fetch.
Fetch 32-bit instruction from L1-cache
- ID = Decode
- 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



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
Some even using AI techniques



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 (can teach whole class on these)

- 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.
- Modern systems often have multiple levels of cache. Usually 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



Pre-fetching

- To avoid misses, data can be pre-fetched, either explicitly by program or by the hardware guessing.
- What are the downsides of pre-fetching?



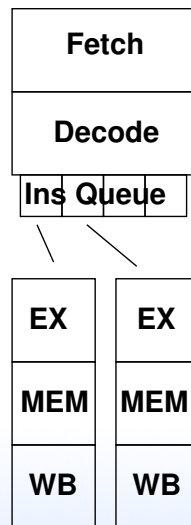
Exploiting Parallelism

- 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?



SIMD / Vector Instructions

- Flynn's Taxonomy
- SISD – single instruction, single data, your normal serial processor
- SIMD – single instruction, multiple data – one instruction can act on many values in parallel
- MISD – multiple instruction, single data – wavefront or pipeline? some debate about if this really exists
- MIMD – sort of like a cluster



Traditional SISD Floating Point

- ARM: series of registers (separate from general purpose)
Use special instructions like fadd, fsub, fmul
- x86: The 8087 floating point is very complex
stack based, 80-bit floating point



SIMD / Vector Instructions

- x86: MMX/SSE/SSE2/AVX/AVX2
semi-related FMA
- MMX (mostly deprecated), AMD's 3DNow!
(deprecated)
- PowerPC AltiVec
- ARM: Neon / A64 SIMD / “Scalable Vector Extensions”
SVE



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 instructions



AVX-512

- now 512 bits, ZMM0-ZMM31
- Originally developed on discontinued Xeon Phi architecture
- Skylake and newer x86 (similar but not same as on Phi), but not always on desktop and they disabled with update on hybrid systems
- Issues with throttling due to heat slowing down too much
- AMD caught up also offers



SSE example (From Wikipedia)

Doing a 4 element single-precision vector add would take 4 separate floating point adds:

```
vec_res.x = v1.x + v2.x;  
vec_res.y = v1.y + v2.y;  
vec_res.z = v1.z + v2.z;  
vec_res.w = v1.w + v2.w;
```

With SSE you only need one add instruction:

```
; move aligned packed single precision  
movaps xmm0, [v1]           ; xmm0 = v1.w | v1.z | v1.y | v1.x
```




```
addps xmm0, [v2]
movaps [vec_res], xmm0
```

```
; xmm0 = v1.w+v2.w | v1.z+v2.z
```



Intrinsics

TODO: make this match the previous example

```
__m256i in1;
```

```
/* vmovdqa (%rcx),%ymm1 */
```

```
__m256i filter_avx = _mm256_load_si256( (__m256i
```



ARM NEON

- Cortex A8, optional on Cortex A9
- 64 or 128bit, but some procs break 128-bit into two operations
- 8, 16, 32-bit ints, single-precision floating point



ARM Scalable Vector Extension (SVE)

- Scale from 128 to 2048 bits transparently
- 32 scalable registers Z0-Z31, bottom 128 bits V0-V31
NEON registers
16 predicate registers P0-P15, fault register
- int/double/float/half
- There are intrinsics
- Can also try -fvectorize with compiler
- SVE2 adds more instructions beyond HPC workloads
UDOT – for machine learning



TBL and TBX – computer vision

CADD and CMLA – baseband networking

BDEP and BEXT – genomics

MATCH and NMATCH – server



RISC-V vector extension

- TODO: look up info on this



SIMD Benefits

- Can be faster (2, 4, 8, 16, etc. things at once)



SIMD Drawbacks

- Harder to code (assembly or clever compiler)
- Puts more pressure on memory.
- More registers to save at context switch

