Condensed Comp Arch Notes

**Amdahl’s Law:** Benefits of speedup are limited by the non-sped up parts

* Time = unoptimized part + optimized/speedup
* Memory bytes don’t have intrinsic meaning, their meaning depends on context
* Little endian format: highest address goes in the most significant position

**Compilation files**

* .c file has C code
* .s file has assembly code with assembly commands
* .o file has x86 machine code converted from assembly
  + Have a place holder for address accesses that we’re not sure about the location yet
  + The assembler then holds extra information for these addresses called relocations which say at byte whatever, the placeholder needs to be replaced with the actual address (when you find it at link time)
  + Has a symbol table which says where headers (like main) start
* .o files are linked together to produce an executable
* \*string data ASCII is in every file

**Pointer arithmetic**

* Arrays decay as pointers (to the first element in the array)
  + \*(foo + bar) is the same as foo[bar]
* Sizeof(array) gives the size of all elements while sizeof(pointer) is the size of the address
* Goto in C functions exactly as an assembly jump

**Structs**

* Typedef struct [structname] upon declaration of a struct allows you to give it a name to use without the struct preceding it
* Not references, they’re instances with their own storage so they are not stored on the heap

**Undefined Behavior**

* Common types: signed integer underflow/overflow, out-of-bounds pointers, integer divide by 0

**AT&T Syntax**

* q = 8 bytes, l = 4 bytes, w = 2 bytes, b = 1 byte
* memory addresses: movq $42, 100(%rbx,%rcx,4) this is 100+rbx+4\*rcx
* labels represent addresses (of from label: in assembly)
* LEA:
  + Doesn’t access memory
  + It’s a good trick for multiplication leaq (%rax,%rax,4) = 5\*rax

**Condition Codes**

* Set by almost all arithmetic instructions
* Cmp is just a subtraction without remembering results
* \*result = 0 means equal; result = positive means greater; result = negative means less than
* Flags:
  + ZF(zero flag) -> was result 0?
  + SF(sign flag) -> was result negative?
  + CF(carry flag) -> did computation overflow? unsigned
  + OF(overflow flag) -> overflow? Signed

**Microarchitecture vs. Instruction Set**

* Microarchitecture is the design of the hardware
* Instruction set is the interface visible by software, there are many ways to implement it
* \*different microarchitectures can implement the same instruction set

**Instruction set design**

* Variation in:
  + Instruction length
  + # normal registers
  + Approximate # of instructions
  + Condition codes
  + Addressing modes (ways of specifying operands)
  + #of operands
  + Instruction complexity
* RISC:
  + Why? Complex instructions aren’t faster and are harder to implement
  + Fewer, simpler instructions (Y86 sort of has this)
  + Separate instructions to access memory (Y86 has this)
  + Fixed-length instructions (Y86 doesn’t have this)
  + More registers
  + No instruction with 2 memory operands
  + Few addressing modes (Y86 has this)

**And**

* Bitwise and with 1 -> keeps the bit that was already there
* Bitwise and with 0 -> clears bit

**Or**

* An or mask is whatever bits you want to be set no matter what they were before

**Xor**

* Flips the bits

**Hardware**

* Registers update every clock cycle
* Memory is always outputting a value, but writing happens on the rising edge of the clock
* ALU is not dependent on the clock
* MUX is an if then in hardware

**CPU**

* The in class processor is one cycle per instruction. Calculations happen between the rising edges of the clock, rising edge signals state change
* In memory stage: tricky cases for address are popq and ret where the input is not the ALU output
  + Tricky cases for write data are call and push where it’s not valB

**Hclrs**

* Value[2..9] 2 is inclusive and 9 is exclusive

**Pipelining**

* General idea: you can get more done if you’re doing more than one thing at once
* Latency: amount of time it takes to get one thing done
* Throughput: rate at which we get many things done (one way to measure is how long you wait between each time you finish a task
* Use registers to hold intermediate values
* Pipelining speedup is limited by register delay because there is some amount of time before the rising edge of the clock where input must be stable
* Throughput calculation: 1/(register delay + slowest operation time)
  + There’s diminishing returns because of register delays so eventually you put in a lot of effort for not that much gain (most gain from splitting it up the first few times)
  + slowest path through pipelined cpu is called the **critical path**

**Hazards**

* hazard: do the straightforward thing and get the wrong answer
* data hazard is when you need the newly computed value and the old value is still in the register file when you decode
* one solution: stalling
  + hardware inserts nops until we have the value ready that we need
  + requires extra logic
  + bubbles are the nop instructions sent through the pipeline
  + can ALWAYS fix data hazards
* control hazard is when you don’t know what instruction you need to run next (with conditional instructions)
  + also happens with ret
* stalling is expensive and if hazards are everywhere it can actually end up being slower than nonpipelined
* another solution: forwarding
  + forward values if the source to the register file is the same as the writeback location in another stage that’s not yet written back
  + doesn’t work for the case where you load a value and then immediately try to do computation on it -> in this case you must stall one cycle and then forward
* ret requires 3 cycles of stalling no matter what
* longer pipeline will make hazards worse and stalling worse
* shorter pipeline will reduce hazards but a cycle takes longer

**Caching**

* Most of the space in processors is devoted to caches
* Locality:
  + Temporal: access something now, I’m going to access it again soon
  + Spatial: access something now, use something near it soon
  + These are natural properties of programs that help with cache performance
* Set: row in cache:
* Address contains:
  + Tag: part of address not used in mapping; shows where it came from in memory
    - Any part of the address not used in index and offset
  + Index: which row in the cache might contain the information for the address
    - Depends on number of sets
  + Offset: tells you which byte in the data in the cache is being addressed
    - Size of offset depends on the number of bytes in each block
  + **Format:** tag\_\_\_index\_\_\_offset\_\_\_