CS 33 Cheat Sheet

Bits — about operations on numbers

{Performance}

Amdahl's Law (overall speed improvement) — how much speed improvement for computer as a whole from improvement (speedup) M of that component that uses fraction F

- $\text{speedup} = \frac{1}{(1-F) + \frac{F}{M}}$
- $\text{performance} = \frac{1}{\text{execution time}}$
- $\text{speedup} = \frac{\text{performance}_{\text{new}}}{\text{performance}_{\text{old}}} = \frac{\text{execution time}_{\text{old}}}{\text{execution time}_{\text{new}}}$
- $\text{CPU time} = \frac{\text{seconds}}{\text{program}} = \frac{\text{instructions}}{\text{program}} \times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{cycle}}$
- $\text{CPI} = \frac{\text{cycles}}{\text{instruction}} = \frac{\text{CPU time} \times \text{clock rate}}{\text{instruction count}}$
- $\text{MIPS} = \frac{\text{instruction count}}{\text{execution time} \times 10^6} = \frac{\text{clock rate}}{\text{CPI} \times 10^6}$

CPI: (clock cycles per instruction) — avg clock cycles each instruction takes

- $\text{CPI} = \frac{\text{clock cycles}}{\text{instruction count}}$

MIPS: (million instructions per second)

- $\text{MIPS} = \frac{\text{instruction count}}{\text{execution time} \times 10^6} = \frac{\text{clock rate}}{\text{CPI} \times 10^6}$

BANDWIDTH: (bits/sec)

- $\text{bandwidth} = \frac{\text{data size}}{\text{data transfer time}}$ — data size: amount of data transfered; data transfer time: amount of time taken

{Cc Conv}

- $\text{cos}\,\theta = \frac{\text{adj}}{\text{hyp}}$
- $\text{sin}\,\theta = \frac{\text{opp}}{\text{hyp}}$
- $\text{tan}\,\theta = \frac{\text{opp}}{\text{adj}} = \frac{\text{sin}\,\theta}{\text{cos}\,\theta}$
- $\text{hyp} = \sqrt{\text{opp}^2 + \text{adj}^2}$

{Power}

- $\frac{P_2}{P_1} = \frac{(V_2)^2 f_2}{(V_1)^2 f_1}$ — voltage
- $P_2 = \frac{V}{V_1} \cdot P_1$ — frequency
- $P_{\text{dynamic}} = \text{capacitance} \times \text{voltage}^2 \times \text{frequency}$
- $P_{\text{static}} = \text{current} \times \text{voltage}$

{Instruction Set Architecture}

- mem: 64-bit, 8-byte
- 8 rips, 8 rflags
- 16 r0 - r15
- integer convention (RDI, RSI, RDX, RCX, R8, R9) — ints, pointers of function arguments are passed in the order shown

- Byte Ordering:
  - LITTLE Endian: lsb first
  - BIG Endian: msb first
  - 0x0a0b0c0d in memory: 0d 0c 0b 0a / 0a 0b 0c 0d
- 0x0a memory in hyper-addressable: (0f 0e 0d, 00 -> 7f)
- 0x0a0b0c memory in hyper-addressable: (0d 0c 0b, 00 -> 7f)

{Instructions}

- R-type: (register)

opcode	rs	rt	dest	sh	funct

- I-type: (immediate)

opcode (6-12)	rs	dest	im [sext]

- S-type: (stores)

opcode (6-12)	rs	dest	im [sext]	opcode

- SB-type: (conditional branching)

opcode (6-12)	rs	rt	dest	opcode (6-12)	opcode

- U-type: (upper immediate)

opcode (6-12)	d	imm

- UJ-type: (unconditional jump)

opcode (6-12)	d	imm

{Stack}

- push register onto the stack
  - sub rsp, 8
  - mov [rsp], rax
- pop
  - mov rax, [rsp]
  - add rsp, 8
- address calculation from the stack
  - lea r8, [rsp + 16]
  - add r8, [rsp]
- lea r8, [rsp + rax]

- 8 byte aligned

{Memory}

{Tool}

Compiler

Assembler

{I/O}

Write the header ——— read and parse it after open a file

- text segment ——— execute language code
  - r - (R, r-x, RWX) — read, write, execute
- (data) data segment ——— data allocated & work space
  - rw - (R, rw-, RWX)
- allocation information ——— dynamic information/data dependent on execution
  - gdbinfo ——— stack, program is loaded
  - rw - (R, rw-, RWX) — dynamically
- symbol table ——— established table (variables) (functions)
  - rw - (R, rw-, RWX)

- The linker creates this using the executable info and symbol table

{Dynamically Linked Libraries - (DLL)}

- DLLs are not linked and loaded until program is run
- dynamically linked libraries are part of the executable and all symbols are linked into it only when needed
- Lazy Linking:
  - library code is loaded only after it is called
  - GOT (Global Offset Table) / PLT (procedure linkage table)

{Position-Independent Code} (PIC)

- Relative addressing
  - jmp rip, [0x0]
  - jmp rip, [0x8]
  - jmp rip, [0x10]
- The address: (%rip) = 0 + the address address in memory when it is loaded
  - (%rip) = 0x8 — mem_offset
  - (%rip) = 0x10 — mem_offset (0x10)

{Important Operations}

- Subtract:
  - sub (%r0), (%r1), (%r2) — r0 = r1 - r2
  - sub (%r0), (%r1), (%r1,%r2) — r0 = r1 - r2
  - *write* and *rw* address
- Logical:
  - and (%r0), (%r1), (%r2) — r0 = r1 && r2 — ? r0 = r1 > 2^{64}
  - or (%r0), (%r1), (%r1,%r2) — r0 = r1 || r2 — ? r0 = r1 > 2^{64}
  - xor (%r0), (%r1), (%r1,%r2) — r0 = r1 ^ r2 — ? r0 = r1 > 2^{64}
- Loads:
  - lea (%r0), [mem_size+8] — r0 = (mem + 8)000000000000
  - lea (%r0), [%rip][(%r1)] — Memory[r1 + mem] + r10
  - lea (%r0), [%rip][(%r1)] — r0 = Memory[r1 + mem]
- Jumps:
  - jmp (%r0), [mem_size+8] — $\text{Memory}_{\text{r0}}$ + mem + 8
  - jmp* (%r0), [%rip][(%r1)] — $\text{Memory}_{\text{r0}}$ + mem + r1
  - jmp (%r0), [%rip][(%r1)] — $\text{Memory}_{\text{r0}}$ + mem + r1
- Calls:
  - call (%r0), [mem_size+8] — r0 = (mem + 8)000000000000
  - *write address with jmp*
  - *rw address with jmp address*
  - *rw address, or jmpmem then write address to memory, or jmpmem then write to stack*

{Hardware}

- Bits Codes:

- Flow of Execution "Jmp":

(J-Type, not R or or)

(Call)

(ret)

(Long)

{Pipeline}

- 5-stage pipeline
  - IF: get Jmp
  - ID: Jmp decoded and regard the dest
  - EX: ALU, compute a jump address
  - MEM: memory access
  - WB: write back
- Jmp* (%r0), [%rip][(%r1)]

- Jmp* (%r0), [%rip][(%r1)] — jump address is stored in the register %r0 or %r1 or %r1 + %r0
- Jmp (%r0), [%rip][(%r1)] — jump address is stored in the memory
- Jmp* (%r0), [%rip][(%r1)] — jump address is stored in the memory, or jmpmem then write address to memory
- Jmp* (%r0), [%rip][(%r1)] — jump address is stored in the memory, or jmpmem then write address to stack