

Registers

| n | 10 | hex | bin |
|------|----|------|-------|
| \$0 | 0 | 0x00 | 00000 |
| \$at | 1 | 0x01 | 00001 |
| \$v0 | 2 | 0x02 | 00010 |
| \$v1 | 3 | 0x03 | 00011 |
| \$a0 | 4 | 0x04 | 00100 |
| \$a1 | 5 | 0x05 | 00101 |
| \$a2 | 6 | 0x06 | 00110 |
| \$a3 | 7 | 0x07 | 00111 |
| \$t0 | 8 | 0x08 | 01000 |
| \$t1 | 9 | 0x09 | 01001 |
| \$t2 | 10 | 0x0a | 01010 |
| \$t3 | 11 | 0x0b | 01011 |
| \$t4 | 12 | 0x0c | 01100 |
| \$t5 | 13 | 0x0d | 01101 |
| \$t6 | 14 | 0x0e | 01110 |
| \$t7 | 15 | 0x0f | 01111 |

| | | | |
|------|----|------|-------|
| \$s0 | 16 | 0x10 | 10000 |
| \$s1 | 17 | 0x11 | 10001 |
| \$s2 | 18 | 0x12 | 10010 |
| \$s3 | 19 | 0x13 | 10011 |
| \$s4 | 20 | 0x14 | 10100 |
| \$s5 | 21 | 0x15 | 10101 |
| \$s6 | 22 | 0x16 | 10110 |
| \$s7 | 23 | 0x17 | 10111 |
| \$t8 | 24 | 0x18 | 11000 |
| \$t9 | 25 | 0x19 | 11001 |
| \$k0 | 26 | 0x1a | 11010 |
| \$k1 | 27 | 0x1b | 11011 |
| \$gp | 28 | 0x1c | 11100 |
| \$sp | 29 | 0x1d | 11101 |
| \$fp | 30 | 0x1e | 11110 |
| \$ra | 31 | 0x1f | 11111 |

- callee saved registers: \$s0-\$s7, \$sp, \$gp, \$fp
 - * save parent's value at beginning of function
- caller saved registers: basically all the others
 - * save your value before calling subroutine
- general format is to list destination first, then operands

Clock Rate

| period | rate | 10 nsec | 100 MHz |
|----------|--------|----------|---------|
| 1 msec | 1 MHz | 1 nsec | 1 GHz |
| 100 nsec | 10 MHz | 100 psec | 10 GHz |

Metric Prefixes

| | | | |
|-------|-------|------------|-----------------------|
| peta | P | 10^{15} | 1 000 000 000 000 000 |
| tera | T | 10^{12} | 1 000 000 000 000 |
| giga | G | 10^9 | 1 000 000 000 |
| mega | M | 10^6 | 1 000 000 |
| kilo | k | 10^3 | 1 000 |
| hecto | h | 10^2 | 100 |
| deca | da | 10^1 | 10 |
| one | | 10^0 | 1 |
| deci | d | 10^{-1} | 0.1 |
| centi | c | 10^{-2} | 0.01 |
| milli | m | 10^{-3} | 0.001 |
| micro | μ | 10^{-6} | 0.000 001 |
| nano | n | 10^{-9} | 0.000 000 001 |
| pico | p | 10^{-12} | 0.000 000 000 001 |
| femto | f | 10^{-15} | 0.000 000 000 000 001 |

Endianness

Value: 0xA0B0C0D0

| index | 0 | 1 | 2 | 3 |
|--------|------|------|------|------|
| little | 0xD0 | 0xC0 | 0xB0 | 0xA0 |
| big | 0xA0 | 0xB0 | 0xC0 | 0xD0 |

- * Little Endian puts the least significant (littlest) stuff first

- x86 is little endian, MIPS is big endian
- networking is done in big endian

Two's Complement

- N bits can represent a range $[-2^N, +2^N - 1]$
- methods for converting negative values
- method 1:
 - * start with absolute value
 - * flip all bits (bitwise not)
 - * add 1
- method 2:
 - * use $N + 1$ bits (2^N is $N + 1$ bits)
 - * start with absolute value x
 - * find $2^N - x$

* truncate

Shifts

- shift left always fills with 0s
- **Logical** left shift fills with 0s
- **Arithmetic** left shift sign-extends
 - * extends based on far left bit (most significant)

Assembler

- **Spilling**: when a compiler puts a variable in main memory because it's run out of registers
 - * the variable has spilled to RAM
 - * inverse is filling
- **Object file sections**: header; text; data; relocation information; symbol table; debugging information
 - * Object file is assembled assuming that instructions start at 0x00. (this is corrected later by the linker)
- **Global label** can be referenced in any file
 - * you must declare it global in the file where it is defined, and declare it global again where it's used
 - * **main** must be global so the linker can find it
 - * **printf** is global so you can use it (but you must still declare it as global in that file where you use it)
- **local label** can be referenced in only the current file
 - * labels are local by default
- **Symbol Table**: contains all external references
 - * also lists unresolved references (e.g. printf)
 - * as far as assembler is concerned, symbol table contains both local and global labels, resolved and unresolved.
 - * The final assembled object file only contains global labels
- **Relocation Table**: contains references to all things that depend on absolute addresses
 - * e.g. all absolute jumps, load address
 - * these must be changed after loading into memory
 - * does not contain addresses of labels

State Machine

- outputs determined by:
- Mealy Machine: current state and current inputs
- Moore Machine: current state only

Performance

- execution time = (# of clock cycles) \times (clock cycle time) = (# of clock cycles)/(clock rate)
- **CPI**: Cycles Per Instruction
 - * effective CPI is just a weighted average (varies by instruction mix)
- instructions per time = CPI / clock rate = CPI * clock period
- compare two systems:
 - * use instruction latencies and instruction mix to calculate CPI for each setup
 - * then calculate instructions per time, and do comparison there

IEEE Floating-Point

- 1 bit sign; 8 bit exponent; 23 bit mantissa
 - * $x = (-1)^s \cdot (1.m) \cdot 2^{e-127}$
- sign: 0 for positive, 1 for negative
- exponent: bias is -127
- mantissa: the fractional part; denominator 2^{23}
 - * implicit leftmost bit is not stored, only fractional
- conversion: decimal to float:
 - * start with x
 - * use $\lfloor \log_2 \rfloor$ to express x as $a \cdot 2^b$ where $1 \leq a < 2$
 - * exponent = $127 + b$
 - * mantissa = $(a - 1) \cdot 2^{23}$
 - round to nearest integer
- conversion: float to decimal:
 - * real exponent $a = exp - 127$
 - * take exponent as integer $\rightarrow a$
 - * decimal = $(1 + \frac{a}{2^{23}}) \cdot 2^a$

- calculate mantissa directly: $\frac{x}{2^{\lfloor \log_2(x) \rfloor}} \cdot 2^{23}$
- mantissa the long way:
take right-of-decimal part and repeatedly multiply by 2.
On each iteration, the 1's place is that bit in the mantissa. (starting from leftmost bit)
- quantity of numbers on range $[2^n, 2^{n+1}] = 2^{23} + 1$
quantity of numbers on range $[2^n, 2^{n+1}) = 2^{23}$
*the 2^{n+1} bumps it up because the exponent changes
- next largest float: add 2^{-23} to mantissa (assuming exponent doesn't change, i.e. number isn't evenly 2^n)

| exponent | mantissa | meaning |
|----------|----------|--------------|
| 0 | 0 | \pm zero |
| 0 | $\neq 0$ | denormalized |
| 1-254 | any | normal |
| 255 | 0 | $\pm\infty$ |
| 255 | $\neq 0$ | NaN |

| | float | double |
|----------|---------|---------|
| sign | 1 bit | 1 bit |
| exponent | 8 bits | 11 bits |
| exp bias | 127 | 1023 |
| exp min | -126 | -1022 |
| exp max | +127 | +1023 |
| mantissa | 23 bits | 52 bits |

- minimum integer that can't be exactly represented:
 $2^{24} + 1 = 16\ 777\ 217$

Pipelining

- 5 stages: IF, ID, EX, MEM, WB
- branch delay stalls when decision made in stage:
* MEM: 3 stalls; EX: 2; ID: 1
- if a branch is predicted wrong, you must flush the pipeline
- ALU data hazards:
* w/o forwarding: 2 stalls if the next instruction is dependent
* w/forwarding: no delays ever
- load-use hazard:
* w/o forwarding: 2 delays
* w/forwarding: 1 delay for next instruction, then forward from data memory
- ideal speedup: instruction time pipelined / instruction time before = number of stages

Branch Prediction

- if predict wrong, flush the pipeline after the branch
* set all control and opcode to 0 (nop)
- static
- 1-bit: keep a cache of the PC of the branch and last taken/not taken
- 2-bit: keep a cache of the PC of the branch and state in a state machine
* 4 states: strong taken, taken, not taken, strong not taken (ST,T,N,SN)
* taken moves state toward ST, not taken moves toward SN
* allows otherwise consistent branches to have some variation

Cache

- sources of cache miss:
* compulsory: when the cache's valid bit is 0
- cannot be avoided because cache must start empty
* conflict: the wrong data is there
- solve by increasing cache size or associativity
* capacity: when all blocks are full and data isn't there
- really only happens with fully associative, otherwise it would be conflict
- solve by increasing cache size
- locality types:

- * spacial: accessing data that is close to other data
 - * temporal: accessing data that was recently accessed before
 - fully associative
* any data can go anywhere
* tag is full address of data
* must search entire cache to find anything
 - directly mapped (1-way associative)
* use some low-order bits of the memory address to decide where to put the data in the cache (the index)
* means that the tag only needs to be only those bits that aren't in the index (saves space)
* also means that other unrelated data can collide with the existing data by simply having the same low-order bits
 - n-way associative
* same as directly-mapped except there's n total banks of directly mapped cache to check
* (if it's not in one, it might be in another)
* you want to replace stuff in LRU order: Least Recently Used
- thus every cache line must have a counter that is incremented on every cache read
 - multi-block
* combined with above concepts
* store multiple adjacent words from memory together in the cache
* whenever you would fetch just one word in a block, fetch the whole block
* for each word, you have a block offset and an index
* tag is highest order bits of address, then line index, then block index, (then byte offset)
 - write handling:
* naive solution: stall (not optimal)
* write allocate (writeback)
- write into the cache and set dirty bit
- then write to memory when that data is evicted by other data
* no-write allocate
- set dirty bit, but write to write buffer instead
- don't need to stall on writes
- must stall for reads of data that was written to the buffer (stall until buffer is flushed)
 - CPUtime = IC * CPI * CC
* IC: instruction count
* CC: clock cycles
 - CPUtime = IC * (CPI-ideal + Memory-stall cycles) * CC
 - write-buffering:
* read-stall cycles = reads/program * read miss rate * read miss penalty
* write-stall cycles = writes/program * write miss rate * write miss penalty + write buffer stalls
 - write-back: memory-stall = miss rate * miss penalty
- ## Exceptions
- procedure:
* set offending instruction and all following to nop (flush pipeline)
- but keep instructions that came before offending
* set cause and EPC register values
* load (hard-coded?) PC of correct exception handler
* carry on (in exception handler)