

Registers

n	10	hex	bin				
\$0	0	0x00	00000	\$s0	16	0x10	10000
\$at	1	0x01	00001	\$s1	17	0x11	10001
\$v0	2	0x02	00010	\$s2	18	0x12	10010
\$v1	3	0x03	00011	\$s3	19	0x13	10011
\$a0	4	0x04	00100	\$s4	20	0x14	10100
\$a1	5	0x05	00101	\$s5	21	0x15	10101
\$a2	6	0x06	00110	\$s6	22	0x16	10110
\$a3	7	0x07	00111	\$s7	23	0x17	10111
\$t0	8	0x08	01000	\$t8	24	0x18	11000
\$t1	9	0x09	01001	\$t9	25	0x19	11001
\$t2	10	0x0a	01010	\$k0	26	0x1a	11010
\$t3	11	0x0b	01011	\$k1	27	0x1b	11011
\$t4	12	0x0c	01100	\$gp	28	0x1c	11100
\$t5	13	0x0d	01101	\$sp	29	0x1d	11101
\$t6	14	0x0e	01110	\$fp	30	0x1e	11110
\$t7	15	0x0f	01111	\$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		
1 msec	1 MHz	2 nsec	500 MHz
100 nsec	10 MHz	1 nsec	1 GHz
10 nsec	100 MHz	500 psec	2 GHz
5 nsec	200 MHz	250 psec	4 GHz
		200 psec	5 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

J format (absolute branching)

- cannot change the top 4 bits of PC. (PC[31:28])
- range:
 - * total of 2^{26} instructions or 2^{28} bytes
 - because range is $[0, 2^{26} - 1]$
 - * farthest possible next instruction is 2^{26} away (if PC+4 lies at the beginning of a 2^{28} byte boundary)
 - * worst case is you can only jump 1 instruction ahead (if PC+4 lies at the end of a 2^{28} byte

boundary)

- conversion:
 - * instruction stores 26 bits
 - * right pad with two 0s to get 28
 - * take the top four bits from current PC to get 32
- mask of top 4 bits: 0xF0000000
- **target** = (PC AND 0xF0000000) OR (addr << 2)

Relative Branching

- range: $[PC - 2^{17}, PC + 2^{17} - 4]$
 - * that's in bytes. It's a range of $2^{15} - 1$ words
 - * you lose one from the exponent because it's 2's complement
- conversion
 - * take 16 bit offset, zero pad by 2 (multiply by 4)
 - * add to PC+4 (next PC)
- **target** = (PC + 4) + (addr << 2)
- due to the PC+4 thing, if you want to jump back to the same instruction, the immediate value will be -1

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

Verilog

- always block: synthesize to combinational logic iff:
 - * everything written to is always written exactly once for every case of inputs
 - * the outputs of the always block depend only on inputs that are in the sensitivity list
 - * stuff assigned to inside an always block must be declared `reg`
 - will be optimized out if it's combinational
- bitwise not is `~`
- ternary operator: `cond ? if_true : if_false`
- assignments: `=` is blocking, `<=` is non-blocking
 - * `=`: happens in order
 - * `<=`: happens all at once
- case statement: can use `?` to specify 'don't care' for some bits
- `'timescale unit/precision:`
 - * `unit`: 1, 10, or 100, unit either s, ms, us, ps, fs
 - * `precision`: must be shorter than `unit`

State Machine

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

Performance

- execution time = (# of clock cycles) × (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
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: $2^{24} + 1 = 16\,777\,217$