srai adds signal bits into empty bits

Two Complement Tricks

$X + \sim X + 1 = 0$

MAX_INT + 1 = MIN_INT

-1 = 0x FF...F (all 1's in binary)

$MAX\_INT = 2^{n-1} - 1$, $MIN\_INT = -2^{n-1}$

$\sim X + 1$ to negate

Prologue:

```
addi sp, sp, -12
sw s1, 0(sp)
sw s1, 4(sp)
sw ra, 8(sp)
```

Epilogue:

```
lw ra, 8(sp)
lw s1, 4(sp)
lw s0, 0(sp)
addi sp, sp, 12
```

Branch offset (in bytes)

Jump offset (in bytes)

mod and read from right to left

$2^5 = 32$ $2^6 = 64$ $2^7 = 128$ $2^8 = 256$

$2^9 = 512$ $2^{10} = 1024$

typedef -> don't have to specify struct and can't be overwritten

array var is a pointer to the 0th elem

null terminate strings when copying/mallocing

have to keep org malloc ptr to use free()

cannot return pointer to things on the stack

strings in func should be static or malloced if need to be returned

Another soln is to allocate on the stack in the caller

Callee: ensure saved registers and sp are the same

Caller: must save other registers if needed

```
struct stack * stk = (struct stack *) malloc(sizeof(struct stack));
if (!stk){
    return NULL;
}
stk->size = size;
stk->topIndex = -1;
stk->stackArray = (int *) malloc(sizeof(int) * size);
if (!(stk->stackArray)) {
    free(stk);
    return NULL;
}
return stk;
}
int push(int x, struct stack* stk) {
    if (stk->topIndex == stk->size -1) {
        int *stackArray = (int *) realloc(stk->stackArray,
            stk->size * 2);
        if (!stackArray) { return 0; }
        stk->stackArray = stackArray;
        stk->size = stk->size * 2;
    }
    stk->topIndex++;
    stk->stackArray[stk->topIndex] = x; }
```

a0, a1 -> return value

a0 - a7 -> args

vals to save b4 jal -> a0-a7, t0-t6, ra

vals to restore b4 jalr -> sp, s0-s11

Stack: local var, const (defined in func)

Heap: malloc

Static: static var, global var, string literals, const (not in func)

Code: const (DEFINE), machine instructions

R-type add rd rs1 rs2

I-type addi rd rs1 imm  I-type (load) lw rd, im(rs1)

S-type sw rs2 imm(rs1)

SB-type beq rs1, rs2, label

imm depends on offset (= mul of 4 bytes)

```
Loop: beq x19, x10, End   }1
      addi                }2
      addi                }3
      j Loop              }4
End:
```

4 inst x 4 bytes = 16 = imm

U-type lui x10, 0x87654

used 1i for large imm

or -1 for the upper imm offset

then add the lower bits

UJ-type jal x0, label

or j label

(can reach same with 18 inst away from PC)

Jalr ret = jr ra = jalr x0, ra, 0

To call func at any 32 bit abs address:

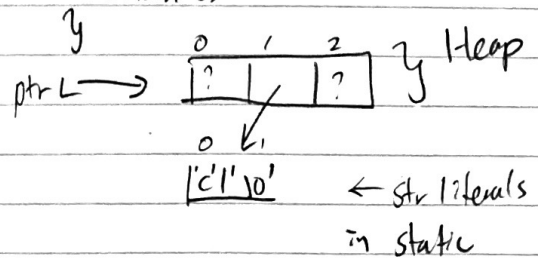
lui x1, <hi 20 bits>

jalr ra, x1, <lo 12 bits>

| Precedence | | Associativity |
|---|---|---|
| 1 | ++ -- suffix | left-to R |
| 2 | ++ -- prefix | Right to L |
| | * | |

```
Packet* read_into_packet(size_t max_size) {
  int size = 0;
  char buf[4];
  Packet* pkt = malloc(sizeof(Packet));
  pkt->data = malloc(sizeof(char) * (max_size));
  while (size < max_size) {
    int n = read4(buf);
    if (!n) break;
    int i = 0;
    while (i < n) {
      pkt->data[size++] = buf[i++];
    }
  }
  pkt->size = size;
  return pkt;
```

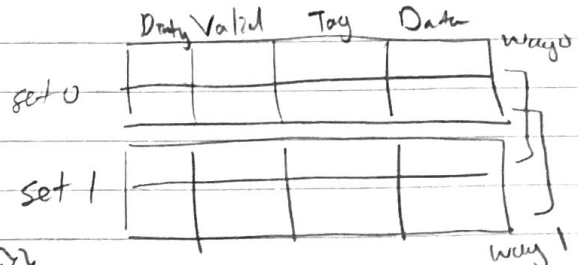
```
int main() {
  char **ptr = malloc(4u);
  ptr[0] = "c";
  printf("%c", ptr[0][0]);
  return 0;
}
```

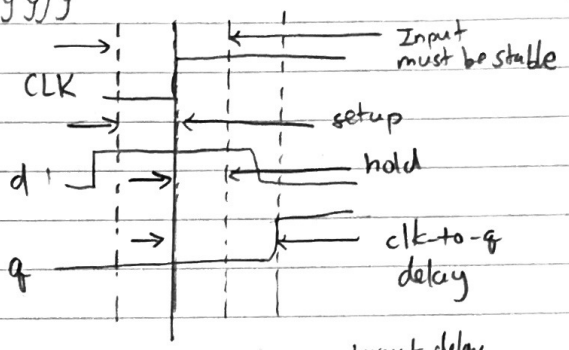



output: c

Jump
Branch (offset in bytes)

```
for (i = 0; i < len(A); i += block) {
  for (j=0; j < len(B); j += 1) {
    for (k = 0; k < block; k++) {
      if (k+i) < len(A) {
        fn(A[i * block + k], b[j]) }}}}
```




Branch Predictor : N-entry, direct-mapped
- when inst is branch check bit for pc[n:2] is set, set next pc to pc + offset (predicted_taken)
when eval, if br taken but pred not, kill
if br not taken but pred taken, kill
Entry starts at 01, if taken add 1, if not sub 1, if upper bit is 1, assume taken

return target location
small stack to check for jalr and jal
when writing to ra, put pc+4 to stack
when reading from ra, pop stack

multiple processor reading OK
Processor A wants to write, write-allocate, set the dirty bit, and broadcast, other processors invalidate the entry if they have it.
Miss in upper level, if no one has it, just fetch
If a processor has it w/ dirty bit set, flushes entry (write to mem), clears the dirty bit and gives the val to the requesting processor
coherence misses, caused by writes by two processors on the same data
Incoherence misses - two processes have diff working sets → reduced cache size and ↑ miss rate
To solve = use virtual mem, great if we never have to fetch from disk

← longest delay

$t_{clk} > t_{clk\text{-}to\text{-}q} + t_{logic} + t_{setup}$

$t_{clk\text{-}to\text{-}q} + t_{logic\,shortest} > t_{hold}$

latency = Time for 1 inst to finish
SS to pipelined, latency ↑
SS : add up delay for all stages
Pipelined : # stages x clock period

Throughput : # of inst processed/second
SS to Pipeline : throughput ↑
# stages / latency

Branch - kill next 2 inst
load - stall 1 cycle
data w/o forward - 2 stalls

Cache Write Hit
Write back, or Write thru

Write Miss
Write allocate - fetch then write
no wa - write to mem and don't fetch

1. How many accesses / loop
   a. count write and read
   a[i] += 5 is read & write
2. How many loops/steps per block
3. How large is cache compared to array
   a. what % of array can fit
   b. Can we reuse cache in later func
4. Do we reuse blocks once we "leave" them
5. How to fix various misses

AMAT = L1 HT + L1 MR (L2 HT + L2 MR * L2 MP)

Global MR - fraction of all accesses that missed at that cache level
   # cache misses / # total accesses

Local MR = Fraction of accesses that occur at that cache level that miss
   Ex. L2 misses / L2 accesses

CL – output as a func of input

Sequential logic

max delay = crit path

## Compiler

In: foo.c Out: foo.s (Assembly lang code)

may contain pseudo-inst (assembler understands but not machine)

Assembler: reads and uses directives

replace pseudo inst, produce Machine lang

creates object file In: foo.s out: foo.o

Assembler directives: give directions to assembler but do not produce machine inst.

.text (machine code) .data (binary rep of data in src file)

.global sym (declares sym global, can be ref'd from other files)

.string str (store the string str in mem & null terminate)

.word w1...wn: store the n 32-bit quantities in successive mem words

Forward ref: jump within a file is okay (PC relative)

branch require two passes

Symbol table: "items" in this file that other files may use (.data)

Relocation table: list of "items" this file needs the address of later

Ex. external jumps or data in static section (la inst)

Object file format: header, text seg, data seg, relocat info, sym table, debugging info.

Linker: In: foo.o, libc.o Out: a.out

Combines object files into executable, enable separate compilation of files. Takes text seg from each, then data seg from each, then resolve abs address

PC Relative (beq, bne, jal) never relocate

External func Ref (usually jal) always relocate

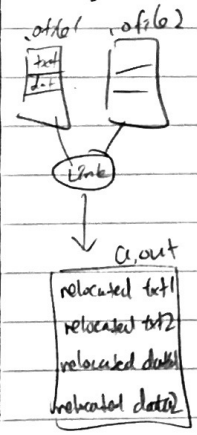
Load/store to var in static area, relative to gp (relocate)

Linker knows len of txt seg and data seg and ordering, calc the abs address of jumps and data referenced

Output: executable w/ text and data (plus header)

Loader: In: executable Out: (program is run)

load to mem from disk and start, usually the OS




Flip-flop – one bit of state, sampled on rising edge

Reg – several bits of state, sampled on rising edge

## Boolean

$1+A=1 \quad A+\bar{A}=1 \quad A+AB=A$

$0B=0 \quad B\bar{B}=0 \quad A+\bar{A}B=A+B$

$(A+B)(A+C)=A+BC$

$\overline{AB}=\bar{A}+\bar{B} \quad \overline{A+B}=\bar{A}\bar{B}$ Demorgan's law

Setup Time: input must be stable b4 edge

Hold Time: input must be stable after edge

Clk-to-Q: how long it takes the output to change after the edge

$t_{hold} <= t_{clk\text{-}to\text{-}q} + t_{shortest\ CL\ from\ any\ reg\ to\ a\ reg\ input}$ (min delay)

$t_{clk\text{-}to\text{-}q} + t_{longest\ CL\ from\ reg\ to\ a\ reg\ input} + t_{setup} <= clk\_period$ (max delay)

## Improving Cache

Reduce time to hit → smaller cache

Reduce miss rate → bigger cache

Reduce miss penalty → cache levels

| | ↑ Associativity | ↑ Entries |
|---|---|---|
| Hit time | ↑ | ↑ |
| Miss rate | ↓ | ↓ |
| Miss penalty | mostly unchanged | Miss penalty unchanged |

Increase in hit time might be worse even if hit rate goes up.

**↑ block size**

Hit time unchanged

Miss rate ↓ then ↑ (more spatial; more conflicts b/c less blocks)

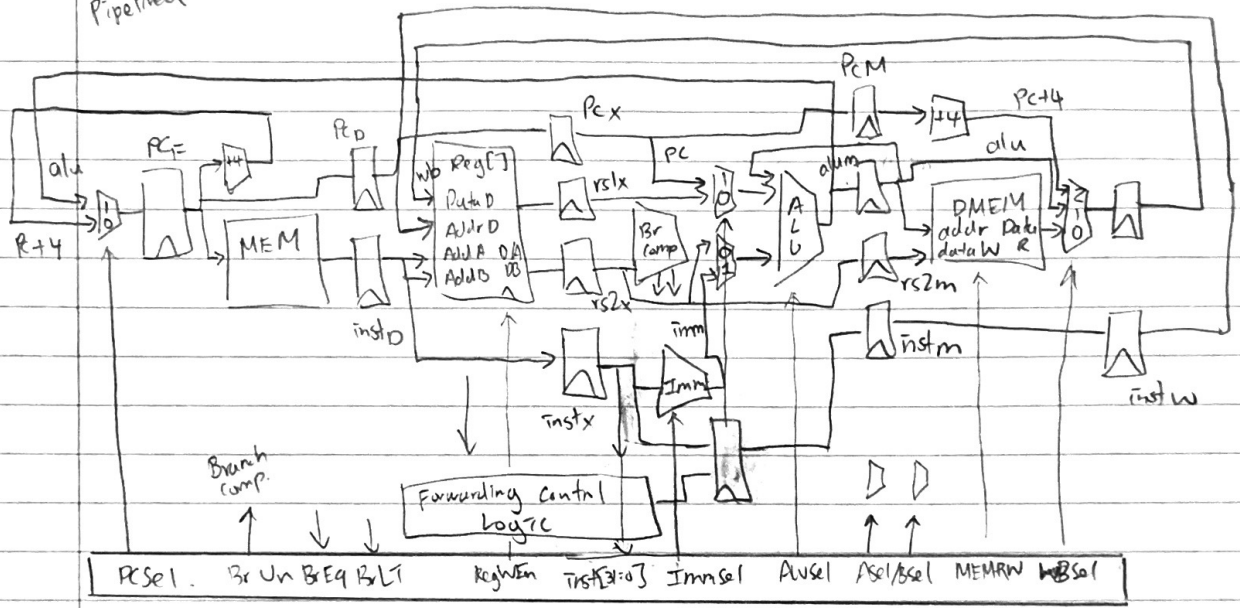
Miss penalty ↑ with ↑ blk size but amortized over whole block w/ fixed const init latency

Global miss rate: misses in this cache / total # of mem accesses by CPU

local miss rate: $LMR_{L2} = L2\ misses / L1\ misses = L2\ misses / total\_L2\_accesses$

L2 local MR >> GMR

Pipelined




lw takes the longest  Processor Performance  $\frac{\text{Time}}{\text{Program}} = \frac{\text{Inst}}{\text{Program}} * \frac{\text{Cycles}}{\text{Inst}} * \frac{\text{Time}}{\text{cycle}}$

Time/cycle = 1/ Frequency

$\frac{\text{Energy}}{\text{Program}} = \frac{\text{Inst}}{\text{Program}} * \frac{\text{Energy}}{\text{Inst}} = \frac{\text{Inst}}{\text{Program}} * CV^2$

Energy Efficiency Law  Performance (Task/sec) = Power * energy efficiency = (Joules/sec) (Tasks/Joule)

Structural hazards: 2 or more inst compete for access to same src
solutions: Stall or add hardware
Ex. IMEM and DMEM
Reg file - sep RW ports

Data hazard: Value not updated before use: Stall or forward

Control hazards: branch & Jump leads to wrong inst fetched

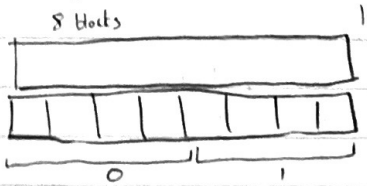
Load data hazard: must stall one inst (load delay slot)

Solution: convert fetched inst to NOPs for next two cycles
branch prediction to reduce performance loss, or put unrelated inst

## Cache

Tag = Addr size - index size - $\log_2$(# bytes/block)  Index = $\log_2$(cache size / block size * N)  =

O = $\log_2$(block size)  $\log_2$(# of sets)




Fully Associative: block can go anywhere (no index)

Direct Mapped: block goes to one place

N-way set associative: N places for a block

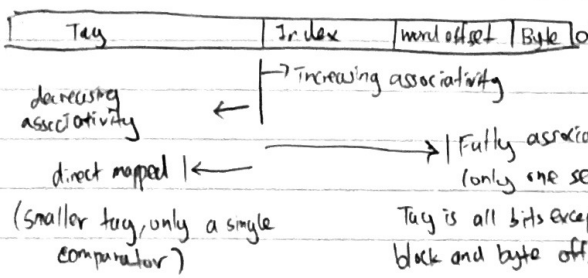
Valid bit: if cache is valid for a matching tag

Write-through: always write to mem

Write-back: write only to cache and write to mem when evicted. (Need a dirty bit)

AMAT = Time for a hit + miss rate x Miss Penalty

Fixed-size cache and fixed block size




## Cache misses

no write allocate: only write to mem

write allocate: fetch into cache

compulsory, conflict, capacity

Prologue
```
addi sp sp -12
sw s0 0(sp)
sw s1 4(sp)
sw s2 8(sp)
```

Epilogue
```
lw ra, 8(sp)
lw s1, 4(sp)
lw s0, 0(sp)
addi sp sp 12
```

Typedef → don't have to specify struct and can't be overwritten

Array var is a pointer to the 0th elem

null terminate string when copying/malloc

have to keep malloc ptr to use free

cannot return pointer to things on the stack

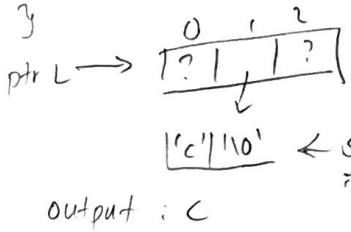
Strings in func should be static or malloced if need to be returned

Another soln is to allocate on the stack in the caller

Callee: ensure saved registers and sp are the same

Caller: must save other registers if needed

```
int main(){
  char **ptr = malloc(40);
  ptr[1] = "c";
  printf("%c", ptr[1][0]);
  return 0;
}
```




output: c

Strlen doesn't include the null terminator

Strcpy includes null terminator

Programed I/O

Not ideal → CPU has to execute transfer, could be doing other work, device speeds don't align well w/ CPU speeds, energy cost of using CPU when it can be replaced w/ simpler hardware

DMA

Allows I/O devices to directly r/w main mem

DMA Engine - contains registers written by CPU

MISC

I=0 for fully associative

Page table base register = # of bits of physical mem

SSDs have finite # of write cycles

Stack: local var, const (defined in func)

Heap: malloc

Static: static var, global var, string literals, const (not in func)

Code: const (DEFINE), machine instructions

a0, a1 → return value

a0-a7, → args

vals to save b4 jals → a0-a7, t0-t6, ra

vals to restore b4 jals → sp, s0-s11

R-type add rd rs1 rs2

I-type addi rd rs1 imm

S-type sw rs2 imm(rs1)

SB-type beq rs1, rs2, label

imm depends on offset (mul of 4 bytes)

UJ-type jal x0, label

or j label

can reach from w/in $2^8$ inst from PC

jalr ret = jr ra = jalr x0, ra, 0

To call func at any 32 bits abs address:

```
lui x1, <hi20bits>
jalr ra, x1, <lo12bits>
```

Map Reduce

map → returns newly distributed datasets

flatMap → similar to map but each input can be mapped to 0 or more outputs

reduceByKey → returns (K, V) pairs where each key are aggregated using some reduce func

reduce → Agg regardless of key using a func

load reserved
lr rd, rs1
load the word pointed to by rs into rd and add reservation

store conditional sc rd, rs1, rs2
store value in rs2 into mem location pointed by rs1 if the reservation is still valid and write status in rd

0 if success
non zero → failure, updated by others, write doesn't happen

```
Loop beq x19 x10, end
  addi
  addi
  j loop
end:
```

2
3
4

4 inst x 4 bytes = 16 = imm

I-type (load) lw rd imm(rs1)

U-type lui x10, 0x87654

used if for large imm or -1 for the imm offset then add the lower bits

More floating

largest finite pos value $(2-2^{-23}) \times 2^{127}$

smallest normalized $2^{-126}$

step size = $2^{-23+n}$ (n = x-127)

Polling → forces hw to wait for ready bit, low latency, good when device always busy or can't move on until device replies, can't do anything while polling

Interrupts → hw raise exception when ready. CPU changes PC to execute code in the interrupt handler. can do work while waiting, can work on more than 1 device, good for devices that take long to respond. Non-deterministic when interrupt occurs. some overhead

| Exponent | Significand | Meaning |
|---|---|---|
| 0 | Anything | Denorm |
| 1-254 | Anything | Normal |
| 255 | 0 | Infinity |
| 255 | nonzero | NaN |

smallest rep pos # $2^{-149}$ (i.e., $2^{-126} \times 2^{-23}$)
second smallest $2^{-148}$ (i.e., $2^{-126} \times 2^{-22}$)

Modified - up-to-date, dirty, no other cache has a copy, can write, mem out of date
Owned - up-to-date, other caches may have copy but must be in shared state)
Shared - up-to-date, other caches may have a copy
Exclusive: up-to-date, no other cache has copy, ok to write, memory up to date
Invalid = not in cache

Polling

$$\frac{\text{clocks}}{\text{sec}} = \frac{\text{polls}}{\text{sec}} * \frac{\text{clocks}}{\text{poll}}$$

$$\%\ \text{processor for polling} = \frac{}{\text{cpu[clocks/sec]}}$$

Interrupt - caused by an event external to curr program running (key press)
Exception - caused by event during execution of inst of curr running program (div by 0)
Trap - action of servicing interrupt or exception by hardware jump to "interrupt or trap handler" code
Interrupt is efficient for low rate

Availability = MTTF/(MTTF + MTTR) in %
Annualized Fail Rate $\frac{(1000 \text{ disks} * 8760 \text{ hrs/year})}{100{,}000 \text{ hrs}} = 87.6/\text{year}$

Latency - Time to do one task
Bandwidth - Tasks finished per unit time

$$\text{Time} = \frac{\text{seconds}}{\text{program}}$$

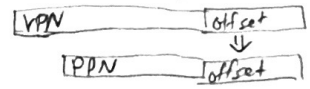
$$= \frac{\text{Inst}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Inst}} \times \frac{\text{seconds}}{\text{clock cycle}}$$

↑ total # of inst executed to run program
↑ Avg # of cycles/inst
↑ sys clock period (1/F)

$$PUE = \frac{\text{total building power}}{\text{IT equipment power}}$$

$$\text{Energy Efficiency} = \frac{\text{Utilization}}{\text{Power}}$$

Virtual Memory




PTE - 1 Valid Bit
2 R/W Bits + PPN or DPN
1 Dirty Bit

Page Table Base Register
↳ points at start of PT

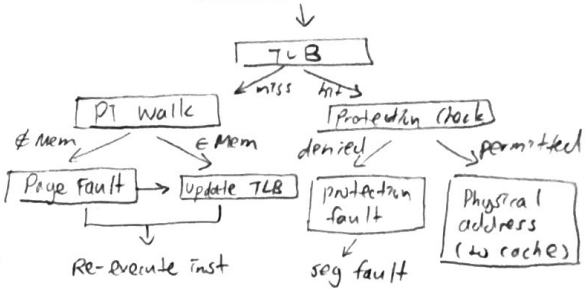
Amdahl's Law

$$\text{Speedup} = \frac{1}{(1-F) + \frac{F}{S}}$$

← speed-up part
← accelerated factor
run speed up ↑

OMP - shared memory
pragma omp parallel for
↳ divide index per thread sequentially (no false sharing)
reduction - specifies 1 or more private var that combines at the end
func: (+, *, -, min, max, &, &&, |, ||, ^)
critical ⇒ locks, only 1 thread runs at a time
Each hardware thread has its own cache

Virtual Address




Disk Access Time
= Seek time + Rotation Time + Transfer Time + Controller
Seek time = # of tracks / 3 * time to move across one track
rotation Time = ½ time of a rev
Transfer Time = size/transfer rate
Flash has no seek time

ECC
bits can cover = $2^n - 1 - n$
Incorrect bit = . sum(parity bit label)
Eg P2 and P8 in error
↳ position 10 is wrong

| Bit position | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Encoded data | p1 | p2 | d1 | p4 | d2 | d3 | d4 | p8 | d5 | d6 | d7 | d8 | d9 | d10 | d11 |
| Coverage p1 | X | | X | | X | | X | | X | | X | | X | | X |
| p2 | | X | X | | | X | X | | | X | X | | | X | X |
| p4 | | | | X | X | X | X | | | | | X | X | X | X |
| p8 | | | | | | | | X | X | X | X | X | X | X | X |

RAID 0
↳ Spread data across disks, bandwidth ↑ linearly, doesn't help w/ latency

RAID 1
↳ Mirror the data, writes limited by single disk speed, expensive, reads may be optimized

RAID 3 ← byte-level
↳ striping w/ parity disk, subtract P from sum to fix err
good for sequential reads/writes

RAID 4
↳ striping w/ parity disk on the block level, good for random reads, bad random writes (need to write to each P disk), good for small reads

RAID 5
↳ Striping w/ interleaved parity, independent writes possible,
↳ Write Algo: Read old data, parity → write new data & parity
↳ High IIO rates

Two's Complement Tricks
$X + \sim X + 1 = 0$
MAX_INT + 1 = MIN_INT
$-1 = 0xFF\ldots F$
$MAX\_INT = 2^{n-1} - 1$, $MIN\_INT = -2^{n-1}$
$\sim X + 1$ to negate
$2^5 = 32$ $2^6 = 64$ $2^7 = 128$
$2^8 = 256$ $2^9 = 512$ $2^{10} = 1024$

Branch offset (in bytes)
Jump offset (in bytes)

Divide by base and continue w/ remainder until 0, read remainders from last to first