ISA

Components: processor, I/O, mem, and network.

ISA: formal specification of the instruction set that is implemented in the machine hardware.

Simplicity favours regularity; Smaller faster; Make common case fast; Good design demands good compromises;

Important registers:

(ra: usually PC + 4); (sp: must be aligned to 4 bytes); (gp: holds the base address of global variables)

Arithmetic

rs1 and rs2 fields need to be kept in the same place: imm field in S-type separated

`destination = source1 op source2`

In I format, values range: -2^{11} to $2^{11}-1$.

Load 32 bits:

```
lui t0, 1010 1010 1010 1010 1010b
ori t0, t0, 1010 1010 1010b
```

logical shift: fill the vacancy with zeros

```
slli t2, s0, 8   # t2 = s0 << 8 bits
srli t2, s0, 8   # t2 = s0 >> 8 bits
lw t0, 4(s3)   # load word from mem to reg
sw t0, 8(s3)   # store word from reg to mem
(loaded or stored using a 5-bit address)
```

Address is byte-base: increment 4 when accessing reg

Little Endian: rightmost byte is the most significant byte.

`lb` places the byte from mem into the rightmost 8 bits of the dest reg and signed extension.

```
lb t0, 1(s3)   # load byte from memory
sb t0, 6(s3)   # store byte to memory
```

stack grows from high address to low address. in recursive procedures, `rd` stored in `ra`, which kept on stack

2's complement: complement all the bits and then add 1

$6 = 00...0110_2 \Rightarrow 11...1001_2 + 1 \Rightarrow 11...1010 = -6$

n-bit signed binary: $[-2^{n-1}, 2^{n-1}-1]$

ALU

32-bit signed numbers: range from $2^{31}-1$ to -2^{31}

If the bit string represents address: 0 to $2^{32}-1$.

Sign extension copies the most significant bit into the other bits to preserve the sign of the number.

Ripple Carry Adder: connect all adders in sequence, slow because each bit's carry-out depends on the previous bit's carry-in, leading to a cumulative delay (glitching).

Glitching: invalid and unpredictable output that can be read by the next stage, resulting in incorrect behavior.

Critical path: longest sequence of dependent operations

Overflow: adding two positive numbers yields a negative / adding two negative numbers gives a positive / subtracting a negative from a positive gives a negative / subtracting a positive from a negative gives a positive.

To detect overflow, use `carry-in MSB ^ carry-out MSB`

n-bit $\times$ m-bit multiplication, must have $n+m$ bits to cover all possible products.

`mul`: 32-bit $\times$ 32-bit multiplication and places the lower 32 bits in the destination register. `mulh`, `mulhu`, and `mulhsu` perform the same multiplication but return the upper 32 bits of the full 64-bit product.

Logical shifts fill with zeros, while arithmetic right shifts fill with the sign bit.

Floating $\underbrace{6.6254}_{\text{Mantissa (always +)}} \times \underbrace{10}_{\text{Base}}{}^{-27} \Longleftarrow \pm 1.M \times 2^{E'-127}$

Structure: S — E' — M (1 - 8 - 23)

S: Sign bit; E': 8-bit signed exponent; M: mantissa

e.g. $\texttt{40C0000}_{16}$ in decimal

1. 40C0000 = `0 10000001 10000000000000000000000`
2. Sign bit (0): Positive (+)
3. Exponent: $10000001_2 - 127 = 129 - 127 = 2$
4. Mantissa: $1.1000000000... = 1 + 1 \times 2^{-1} = 1.5$

Result: $1.5 \times 2^2 = 6$

e.g. -0.5_{10} in binary

1. Sign bit: 1
2. Mantissa: $0.5 = 1.0 \times 2^{-1}$
3. Exponent: $127 - 1 = 126 = 01111110$

Result: -0.5_{10} = `10111111000000000000000000000000`

$E = 0, M = 0$: 0; $E = 0, M \neq 0$: denormalized number, which is $\pm 0.M \times 2^{-126}$; $E = 1\ldots1$, $M = 0$: $\pm\infty$, depending on the sign; $E = 1\ldots1$, $M \neq 0$: `NaN` (Not a Number).

Datapath

combinational: ALU; state: memory

The partition of `imm` field: align the `imm` with other instruction types - more efficient implementation of control units.

The instruction is decoded in the path between the Instruction Memory and Register File.

Pipeline

clock cycle: timed to accommodate the slowest instruction: instr take same amount of time

CPU time = CPI$\times$CC$\times$IC, CPI = cycles per instruction, CC = clock cycle time, IC = instruction count.

IF: Instruction fetch and PC update

ID: Instruction decode and register file read

EXE: Execution or address calculation

MEM: Data memory access (only in loads and stores)

WB: Write the result data back into the register file

instruction latency (time from start of instr to completion) is not reduced.

State registers between pipeline stage: flip-flop, data moves in at rising edge

Structural hazards: conflicts in the use of a resource

- separating instruction and data memories
- reads in the 2nd half of the cycle and writes in the 1st half

$$\text{Clock rate} = \frac{1}{\text{Clock cycle time}}$$

Data Hazards: dependence of one instr on an earlier one

- Read After Write (RAW)
- Load-Use data hazard

Solution: 1. Insert NOP / Stall 2. Forwarding

Control Hazards: make decision based on results of 1 instr

1. Unconditional branches: `jal`, `jalr`

2. Conditional branches: `beq`, `bne`
3. Exceptions
Solution: 1. Stall
2. delay branching (move branch hardware to ID stage, use delay slot to do other instr first)
3. branch prediction
`NOP` is determined at the compilation stage, while a flush is determined during runtime
- static branch prediction: assume not taken (works for top of the loop, not for btm of the loop); assume branch taken need one stall cycle
- dynamic branch prediction: use run-time info (branch prediction buffer), if prediction wrong, flush the incorrect instr, restart pipeline

Exception: Traps - division by zero (synchronous); Interrupts - pressing keyboard while compiling (asynchronous)

Performance

Response Time: start and completion of a task
Throughput: total amount of work done in a given time

$$\text{performance}_X = \frac{1}{\text{execution time}_X}$$

If X is n times faster than Y, then

$$\frac{\text{performance}_X}{\text{performance}_Y} = \frac{\text{execution time}_Y}{\text{execution time}_X} = n$$

CPU exe time = # of CPU clock cycles×clock cycle time

$$= \frac{\text{number of CPU clock cycles}}{\text{clock rate}}$$

CPU clock cycles = # of instr × clock cycles per instr

$$\text{CPI}_{\text{eff}} = \sum_{i=1}^{n} \text{CPI}_i \times \text{IC}_i$$

IC_i: percentage of the no of instructions; CPI_i: ave no of clock cycles per instruction for that instruction class.

CPU time = Instruction count × CPI × clock cycle time

$$\text{GM} = n \cdot \sqrt{\sum_{i=1}^{n} \text{SPEC ratio}_i}$$

Memory

$1\,\text{k} \approx 2^{10}$ (kilo) 1000; $1\,\text{M} \approx 2^{20}$ (Mega) 1000000;
$1\,\text{G} \approx 2^{30}$ 1000000000; $1\,\text{T} \approx 2^{40}$ (Tera) 1000000000000
Small memories are fast, large memories are slow.
L1: Cache using Static RAM (SRAM)
L2: Another cache, using faster Dynamic RAM (DRAM)
L3: Main memory, typically DRAM
L4: Secondary memory such as flash storage or solid-state drives (SSD)
first three: volatile - retain data when power on
Temporal Locality (time), if referenced, will be referenced again soon: keeps the most recently accessed data closer to processor; Spatial (space - array), location is referenced, locations with nearby addresses will be referenced soon
RAM: SRAM, DRAM, SDRAM, and DDR SDRAM
SRAM: ≥ 6 transistors, fast, large, expensive
DRAM: 1 transistor, capacitor, slow, small, cheap
Interleaving: hide memory access latency
Secondary memory: e.g. magnetic disk
Reg - Cache (SRAM) - Main Mem (DRAM) - Disk - Tape

Cache

Direct mapping: tag (identify which block from MM) + blk no (index the cache) + byte address
Direct mapped cache with 2^n blocks, n bits are used for the index. For a block size of 2^m words (2^{m+2} bytes), m bits are used to address the word within the block.
2 bits are used to address the byte within the word.
Tag size= 32 - (n + m + 2)
The total number of bits in a direct-mapped cache is:
$2^n \times$ (block size + tag field size + valid field size)
1 word = 4 bytes = 32 bits (diff bit hv diff calculation)
Associative Mapping: arbitrary block
Combine: use tag + set + byte
cache write hit: write-through: write both mem and cache (use buffer); write-back: only write cache and write to mem when cache is replaced (dirty bit)
Asso replace: least recently used / random replace
Average Memory Access Time = $h \times C + (1-h) \times M$
h - hit rate, C - cache access time, M - miss penalty
Miss Penalty: total access time experienced by the processor when a miss occurs (1 + X + 1)
high-performance, C_1: cache access time, C_2: miss penalty from L2 to L1, M is from main memory to L2 to L1:
$h_1 \times C_1 + (1-h_1) \times [h_2 \times C_2 + (1-h_2) \times M]$

Virtual Memory

Address space: pages (fixed size) or segments (variable sizes). Frequently used blocks are copied into the cache.
page table: stores mapping between virtual and physical pages. recent translations may be cached in the Translation Lookaside Buffer for faster access.

Instruction Level Parallelism

Multiple-Issue: instructions per cycle (IPC)
static multiple-issue processor (VLIW): compiler decides which instructions can run in parallel; dynamic multiple-issue processor (superscalar): CPU decides at runtime
Data hazards: True Dependency (RAW); Anti-dependency (WAR); Output Dependency (WAW).
Register renaming: rename original reg to a new reg
VLIW: (1) instruction scheduling: must separate load-use instrs from their loads by one cycle, avoid stalls, and (2) loop unrolling (copy the loop body to reduce loop)
Superscalar: Dynamic multiple-issue processors decide at runtime which instrs to run in parallel