1. Abstraction (Layers of representation/interpretation)
2. Moore's Law (Designing through trends)
[2x transistors / chip every 2 years]
3. Principle of Locality (Memory Hierarchy)
4. Parallelism [requests / threads / insts / data]
5. Performance Measurement and Improvement
6. Dependability via Redundancy

Number representation: $2^6=64$, $2^8=256$, $2^{10}=1024$

$a_{ten} \to -a_{ten}$ <-> 将 a_{two} 每位取反, then add 1 → $(-a)_{two}$

<=> a_{ten} n bits $(2^n)_{two} - (a_{ten})_{two} = (-a_{ten})_{two}$

overflow: carry into MSB ≠ carry out MSB

	2's complement	1's complement	sign magnitude
01111111	127	127	127
10000000	-128	-127	-0
11111111	-1	-0	-127
00000000	0	+0	+0

$-2^{n-1} \sim 2^{n-1}-1$ (n: 位数 bits); $-(2^{n-1}-1) \sim 2^{n-1}-1$

Decimal to Binary:
① Integer: ÷2, 取余, 余数倒排
② 小数: ×2, 取整, 整数部分顺排, 小数进行下一步×2.

位运算: Get the nth bit: (x>>n)&1
Set the nth bit to 1: x = x|(1<<n) … 0: x = x&(1<<n)
Flip the nth bit: x = x^(1<<n). a xor 0 same, a xor 1 reverse
and FF → LSB, and FF000000 → MSB.
$x<<n = x*2^n$. sra n = $\lfloor x \div 2^n \rfloor$

---

gcc: -Werror: make all warnings into errors.
-Werror=xxx: xxx is the specific type of error
-Wall: enable all warnings about constructions, & some language specific warnings.
-Wextra: enable some extra warning flags that not enabled by Wall.
-Wpedantic/-pedantic: issue all warnings demanded by strict ISO C/C++; reject all warnings that use forbidden extensions
-g/-ggdb: enable debugging mode, add debug infos.
-O0: turn off all the optimizations
-m32: compile it for 32 bits machine
-save-temps: see result of preprocessing

Macro: #str → "str" val##123 → val123
Inline: not guaranteed to be inlined, decide by compilers.
-Os - ignore inline function; GCC try to inline most function if -O2, -O3. recursive doesn't inline
Static doesn't behave well in macro, macro can't access private/protected variables.
#define SQUARE(x)({ typeof(x) _x=(x); _x*_x;}).

Enum a {RED, GREEN, BLUE}
enum a my_a = RED # automatic 0,1,2,3… assign
typedef enum{…} a; a my_a = BLUE # 2.

Integer: int should be integer type that target processor works with most efficiently. only guarantee sizeof:
long long >= long >= int >= short; short >= 16 bits, long >= 32
sizeof execute in compile time: replacement
All could be 64 bits
32 bits machine, pointer, size_t, int size is 4

Boolean: no explicit. False: 0, NULL; True: all that not False.

Operator: a&b, a|b bit wise; a&&b a||b logical.
order: -> > ++ > *. In all, ->, ., () function call, [] highest.

Structure: a=b, not a deep copy. members ✓ things pointed to by pointers ✗.
Pointer: may cause segmentation faults (*NULL) and bus error
[illegible]
C defines that one element past end of array must be a valid address.
int main(int argc, char* argv[]): argc: number of strings. count the path of executable file; char* argv[]: an array of string. argv [illegible]
{ Dangling pointer: point to memory location that has been deleted
{ Wild pointer: (uninitialized pointer) point to some arbitrary memory location.
void incre_ptr(int** h){*h=*h+1}.
char a[]="a message" can't change a. a++ ✗.
char* b="a message". constant string. in static data

C Memory:
void* malloc(size_t n); use headers at start of allocated blocks and space in unallocated memory to hold malloc's internal data structures.
void free(void* p)
void* memcpy(void* dst, const void* src, size_t num);
copy num bytes from location pointed to by src to dst. binary copy. does not check null.
overflows: size of dst & src (array) < num bytes.
Should not overlap. (memmove is option)
void* memmove(void* dest, const void* src, size_t num)
feel like an intermediate exists.
void* realloc(void* ptr, size_t size);
If ptr=NULL, then realloc behaves like malloc.
If size=0, then behaves like free. memory place change.
void* memset(void* ptr, int value, size_t num)
set the first num bytes of the block of memory pointed by ptr to the specified value (interpreted as char)
char* strcpy(char* dest, const char* src);
copy the C string pointed by src to dst, include \0.

ASCII dex 30 - '0' hex41 - 'A' hex61 - 'a'. hex0 - nul

ISA: instruction set Architecture
RISC: Reduced instruction set computing. (simple) Keep IS simple, build fast hardware; let software do complicated ops by composing simple ones
Intel x86, CISC - opposite. (complex ISC)
Assembly operands are registers (Limited #, 32, built directly in hardware)
Standard no-op: add x0, x0, x0. Improve the dissassembler and can potentially improve the processor.
Little endian: Byte3 Byte2 Byte1 Byte0 word address
Addr3 addr2 addr1 Addr0
Big endian: Byte0 Byte1 Byte2 Byte3 word address
Byte needs sign extended: ≥128 i.e. 0X80
Memory align: RISC-V does not required integer to be word aligned.
i.e. 0(x14), 4(x14), 8(x14)… But in practice, aligned access is faster

---

RISC Instr $\pm 2^{19}$ 16 bits - $\pm 2^{18}$ inst. sign-extended.
jal rd, offset. rd=PC+4, PC=PC+{offset<<1} in [1:20]
jal offset(label) == jal x1, offset → pseudo-inst
j offset = jal x0 offset
jalr rd, rs, offset. rd=PC+4, PC=rs+offset
If don't want to record where to jump: jr rs == jalr x0, rs, 0
jr ra function return & jal function_label.
ABI: Application Binary Interface. Define calling convention
When you call another function, that function promises not to overwrite certain registers.
{ Preserved across function call: sp, gp, tp, s0-s11
saver is callee; caller can rely on values being unchanged
{ Not preserved across function call: Argument / return registers a0-a7, ra, temporary registers t0-t6
saver is caller; callee can change them without saving restoring
C { automatic - stack
{ static - variables exist across exits from and entries to procedures

sp → fp
| saved return address |
| saved argument registers |
| saved saved registers |
| local variables |
sp →

← argument on stack (args > 8, complex struct.)
May change frame pointer.
(call frame / call record / procedure frames / activation record.

Basic structure of a function:

```
entry_label: addi sp, sp, -framesize
             sw ra, framesize-4(sp)
             sw …
             ……
             lw ra, framesize-4(sp)
             addi sp, sp, framesize
             jr ra
```

RISC-V convention: global pointer gp points to static
Arguments ≥ 9 → stack.
Same 32 bits instrs used for RV32, RV64, RV128
not t0, t0 == xor t0, t0, -1
RISC-V features, n×16-bit instrs. Extensions to RISC-V base ISA support 16-bit compressed insts and also variable-length insts that are multiples of 16-bits in length.
$\pm 2^{10} \times 32$-bit insts on either side of PC
LUI x10, 0xDEADB; # 0xDEADB000 pre ++ lui. → 进位 0xDEADC
ADDI x10, x10, 0xEEF # x10 = 0xDEADAEEF (-273) li x10, 0x… works
Branch & jal ×2. 其他 jalr imm original value
auipc x10, <hi 20 bits>
jalr x0, x1, <lo 20 bits> too big for a jalr

Stack: local variables inside functions, grows downward.
heap: space requested for dynamic data via malloc(); resize dynamically, grows upward.
Static data: variables declared outside functions, doesn't grow or shrink. Loaded when program starts, can be modified.
Code: load when program starts, does not change

Assembler Directives: (.float, .double, .align reserved)
.data store subsequent items in the static segment at the next available address
.text store subsequent instructions in the text segment at the next available address (code segment)
.byte store listed values as 8-bit bytes,
.asciiz store subsequent string in data segment and add null terminator
.word store listed values as unaligned 32-bit words
.globl makes the given label global. label e.g. sym

Environmental Calls:
① Load ID into a0 ② load any arguments into a1-a7.
③ Any return value is stored in argument registers (a_x)

ID	NAME	Description
1	print_int	prints integer in a1
4	print_string	prints the null-terminated string whose address is in a1.
10	exit	exit the program
11	print_character	prints ascii character in a1
34	print_hex	prints hex in a1.

E.g. print string:

```
.data          the name of the data
  str: .asciiz "Hello world"
.text
  li a0, 4
  la a1, str   load the address where str is stored
  ecall
```

(lui / addi)

source:
.word 3 successive
.word 4 memory
.word 3 block.
⋮

RISV-ISA Specification: does not define assembly syntax (e.g. li x10); define binary machine instrs & behaviour
Class: RV32I ✓, RV64I, RV128I } characterized by the width of the integer registers and the corresponding size of address space. ✗ inst size. same (32)
RV32E [illegible]
various extension:
C (compressed): some insts 16 bits. M: integer ×, ÷. V: vector. F: floating D: double Q: quad floating point A: atomic intrs

---

Interpreter: a program that executes other programs. Directly executes a program in the source language
Translator (compile): converts a program from the source language to an equivalent program in another language
How to choose: Interprete machine language → better error message e.g. venus
I: easier to write intepreter; closer to high level; slower but code smaller; provides IS independence
T: efficient; hide source from users.

---

foo.c —compiler→ foo.s —Assembler→ foo.o —linker (+ lib.o)→ a.out (binary) —Loader→ Memory

0. preprocessing (CPP)
foo.c → foo.i. ① replace comments with a single space
② change source program according to commands begin with #. e.g. include, define, if endif.
gcc -E foo.c -o foo.i (-o: output)
let compiler stop after precompiling
1. compiler (CCL)
High-level language → Assembly language may contain pseudo-insts.
gcc -S foo.i -o foo.s
stop after compiling and output assembly program
2. assembly (AS)
① Read and use directives ② replace 伪 insts, assign
③ produce machine language, save in an object file. pc-relative addressing jump

* Pseudo-replacement: mv x1, x2 ⇔ add x1, x2, x0
nop ⇔ addi x0, x0, 0 | not rd, rs ⇔ xori rd, rs, -1
beqz rs, offset ⇔ beq rs, x0, offset
bgt rs, rt, offset ⇔ blt rt, rs, offset
j offset ⇔ jal x0, offset | ret ⇔ jalr x0, x1, offset (0?)
call offset ⇔ auipc x6, offset[31:12]; jalr x1, x6, offset[11:0]
tail offset ⇔ auipc x6, offset[31:12]; jalr x0, x6, offset[11:0].
↑ too far for j.
"Tail Call Optimization": doesn't preserve same ra and jump twice in fun(a){… return foo(y)} alike. It can directly return to where it needs to return to rather than return to where foo is called.

Assembly produce machine code:

① 16 bits RVC: offer shorter 16-bit versions of common 32 bits insts when a. imm/offset is b. one of register is x0 / ABI Link(x1) / ABI stack (x2) c. rd=rs1 d. used registers are the most 8 popular ones.

Assembly can pattern match and turn 32b insts ⟶ 16 b insts.

∴ Only assembler and RISC-V processor itself needs to know the presence of the 16b insts.

② Forward reference e.g. beq … L2 …… L2: …

Taking 2 passes over the program.

First pass rembers position of labels; Second pass uses label positions to generate code.

Internal ✓ jump to other file / stack data ✗

Symbol Table: List of items in this file that may be reference to used by other files. (Labels: function; Data: anything in the .data; global also, variables used access files) (relative address)

Relocation Table: List of items whose address this file need.
(inst address) ① External label jumped to (jalr, jal) include lib files / la insts. e.g. for jalr base registers
(Data address) ② data in static section: (la insts e.g. for lw/sw base registers.

Object File Format:

① Object File Header: size & position of the other pieces of the object file
② Text segment: machine code
③ Data segment: binary representation of the static data in the source file
④ Relocation information: lines of code needs to be fixed up
⑤ symbol table: ✓.
⑥ Debugging infos.

gcc -c test.s -o test.o

3. Linker (LD)

Input: object code files + information tables (precompiled lib.o)

output: executable code 将 .o ⟶ relocated (linker)

① Take text segment from each .o, put together
② Take data …, …, and contenate them onto end of ①
③ resolve reference: go through relocation table and fill in all absolute address. [external jump, lw/sw/… ✓]

* Address:
① PC relative addressing (beq, bne, jal) PIC: position independent code. no need to relocate
② External Function Reference (jal) always relocate
③ Static Data Reference (auipc/addi)
④ Absolute addressing [extern jump, lw, sw to static data]

[Enable separate compilation of files]

4. LOADER (OS)

Input: executable file on disk

Output: load into memory and run start program.

If consider statically, don't choose loader any time

※:

All code bits are generated: always ✗ compilation. no machine code generated

We have all infos needed: 机器code can also be known after compilation: we know.

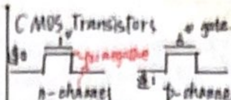
Assembly 后, internal branch ✓. TAL (Typed assembly language)

Linker 后, static ✓ external jump ✓

Static Linking: done by the linker after compiling. The library which is static linked doesn't need to be provided when running the program.

Dynamically linking: done by loader before the program starts or by the program itself after running. Library needs to be provided when running the program.

CMOS Transistors



Low R when gate voltage (gate) low

n-channel {on: v high; off: v low}

p-channel {on: v low; off: v high}

normal of switch

C: complementary: use pairs of normally on and …

Law of Boolean Algebra:

$x\bar{x}=0$ $x+\bar{x}=1$ $x0=0$ $x+1=1$ $x1=1$ $x+0=x$

$xx=x$ $x+x=x$ $xy=yx$ $x+y=y+x$

$(xy)z=x(yz)$ $(x+y)+z=x+(y+z)$

$xy+x=x$ $(x+y)x=x$ $\bar{x}y+x=x+y$ $(\bar{x}+y)x=xy$

$\overline{xy}=\bar{x}+\bar{y}$ $\overline{x+y}=\bar{x}\bar{y}$

{ Combinational logic (CL): ALU ⟹ output is f of input
Sequential Logic (SL): state element, store info

Max Delay = Setup Time + CLQ + CL Delay

Setup Time: when the input must be stable before the edge of the clock

Hold Time: when the input must be stable after the edge of the clock

Clock to Q: how long it takes the output to change, measured from the edge of clk

Flip-Flop: one bit of state that samples every rising edge of the CLK

Register: several bits of state that samples on rising edge of CLK or on LOAD.

Critical path: clk→q + neccessary to get the (divide by regs) output of the regs + worst case CL + setup time for next reg

Hold Time Violation:

clk→q + best case CL < Hold time

don't hold the input to the flip flop long enough

solution: add delay on the best case path

FSM: Finite State Machine

Stage 1: Instruction Fetch
① Fetch inst from memory ② PC=PC+4

Stage 2: Instruction Decode gather data from field lengths.
(①+③ faster than ②: ✗ critical path)
① read opcode to determine inst type & field (generate control signal)
(same time) ② read in data from all neccessary regs.
③ Imm generation (Controller get the inst ✓)

Stage 3: ALU + branch comparison + imm generate.

Stage 4: Memory access: only load & store do things (read & write)

Stage 5: Register write

Store & branch skip this stage

Asserted (1). Negated (deasserted) (0).

Register file 有 1024 bits.

All I-type sign-extend the imm.

PCsel 与 比较相关.

RTL: Register Transfer Level ⊆ HDL (Hardware Description Language)

LW uses all 5 stages (exercise the critical path)

Hazard:

1. Structural hazard: a required resource is busy. (Can always solved by having more hardware.)

① Regfile hazard solution: having seperate ports (2 independent read ports + one indep. write port)

② Memory Access Hazard: use 2 seperate memory

RISCV designed to avoid structural hazards e.g. at most one memory access / inst.

2. Data hazard: data dependency between instrs. (Forward + regfile exploit = restore all.)

① Register Access hazard: exploit high speed of regfile: first half write second half read.

② ALU result hazard: 后面的指令算不到运算结果.

solution (1) Stalling for 2 insts.

Bubble: effective NOP — compiler can do that

solution (2) Forwarding (bypassing): grab operand from pipelined stage, rather than register file

Require connections in the datapath.

Forwarding control logic:

If x.rs == m.rd, select the alu result of m to be an operand of x's EX stage. Must ignore write to x0. ✗ no stall needed then

③ Load Hazard:

Even use forwarding, 1 cycle stall unavoidable

Slot after a load is called a load delay slot.

If that inst use result of load, hardware stall for one cycle.

<solution> code rescheduling, put irrelevant code right after load. compiler / pandem cpu hardware do.

3. Control Hazard

<solution 1> kill the following 2 insts if branch is taken.

<solution 2> Branch prediction. Do that in IF. only flush pipeline if branch prediction was ✗.

single cycle data path: $t_{PC\,clk\to q} + t_{IMEM\,read} + t_{RegFile\,read} + 2t_{mux} + t_{ALU} + t_{DMEM\,read} + t_{RegFile\,setup}$

Pipelined: MEM includes read/write to memory and muxing where to write to 把数据写回.

EX includes branch comp, imm generator, mux and ALU. (same time happen)

Pipelined relative speed < 5✗ reasons:

① add pipeline registers have clk→q and set up times ② need to set the clock to the max of the 5 stages, which usually takes different amount of time. ③ hazard require additional logic to resolve, actual speedup may be even less.

Instruction-Level Parallelism (ILP)

1. Hyperthreading

duplicate all elements that hold the state (reg); use the same CL block (ALU, memory)

2 independent processes, 交替使用资源.

Small / no speedup: 若存在某个stage一方闲置another stage但两者能互相抢占.

2. Multiple issue "superscalar" ways. replicate pipeline stages, start ③ insts per clock cycle. CPI < 1 ∴ use IPC (Instruction per cycle) "

3. Out of order execution

reorder insts dynamically in hardware to reduce impact of hazard.

Iron law of processor performance.

Time per program = insts per program × CPI × Time per cycle

$CPI = \frac{Cycles}{inst} = \frac{Time}{program} \div \left(\frac{insts}{prog} \times \frac{Time}{cycle}\right)$

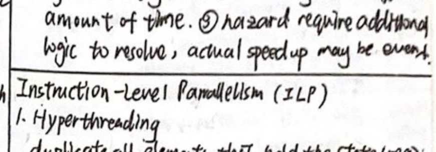
Complex pipeline: Multiple Functional Units ⇒ Parallel execution

1. Static Multiple Issues (Very Long Instruction Word)

e.g. 2 insts per cycle.

compiler takes care of hazard

ALU × 2. Imm gen × 2.



General Purpose Reg GPR
Floating Point Reg FPR

Issue: assign insts to functional unit.

2. Dynamic Multiple Issues.

• Reorder insts. Out of Order OoO

PC → I-cache → Fetch Buffer → Decode / Rename → Issue buffer →

Functional units → result Buffer → commit →

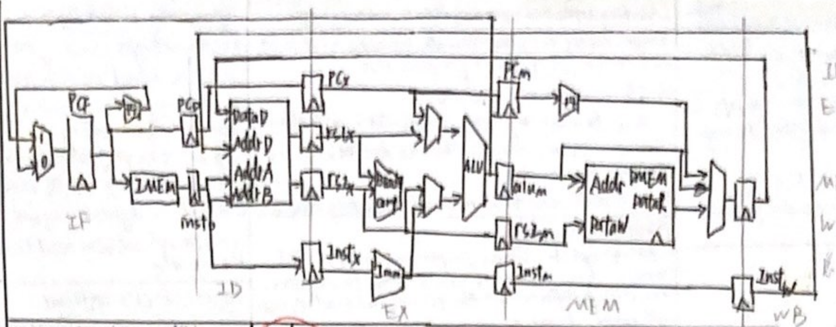
architectural state.

Fetch: inst bits retrived from inst cache

Decode: inst dispatched to appropriate issue buffer

Execute: inst and operands issued to FU. when

二进制	十进制	十六进制
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F



ID: tclk-q + tRFread + tsetup
EX: tclk-q + tmux + tALU + tsetup ? tmem
MEM: tclk-q + tmemread + tmux + tsetup
WB: tclk-q + tRFsetup (? + tRFwrite)
E: (longest) tclk-q + tmemread + tRFread + tmux2 + tALU + tmemread + tsetup

VLIW: (can issue multiple very long inst
- ALU or branch inst — long inst just copy
- Load or store inst
→ combine 2 to 1 large

	out of order superscalar	inorder superscalar	VLIW
Instruction per cycle	<N	≤N	1 large inst. (same work as N normal insts
How do we find INDEPENDENT insts	Look at >>N insts.	Look at next n insts in program order	Just do next large inst
Hardware Cost	Expensive	less Expensive	Even less Expensive
Help from compiler	Compiler can help	Needs help	Completely depend on compiler

Inorder superscalar: CPI ≥ 1. → SISD that supports SIMD.
Fetch 2 insts per cycle; issue both simultaneously if one is integer/memory and the other is floating point (复制 FUs → wider issue)

Out of order superscalar:
Reservation station — Functional units (ALU, Mem) — Commit unit — architectural state (Reg & Mem) same as in order cpu.
waiting 21 27 26 28 | computing 20 19 25 | waiting 22 23 24 | 18 17 16

操作: when execution completes, all results and exception flags are available.
commit: Instruction irrevocably updates architectural state (aka "graduation"), or take precise trap / interrupt.

Seperating completion from commit:
Re-order buffer (ROB) holds register results from completion until commit.
{ Inorder: IF / ID / rename / dispatch — for issue
{ Out of order: EX — put into machine buffers to wait

{ Inorder issue: stalls when dependencies / structural hazard / Read After Write / Write After Read / Write After Write 前面完, 后才可执
{ Out of order issue: Inorder to reservation station, 执行根据 what functional units are free & avoid data hazard.

{ Inorder completion: Classic RISV integer pipeline.
{ out of order completion: ∨ all machine (FU latencies 不同, but desire to dispatch 越快越好)

Register renaming:
Idea: Re-name on the fly to resolve (fake) dependencies.
Additional benefit: CPU can have more physical regs than 32
Alternative to Data-in-ROB: Unified Physical Register File
When next writer of same architectural register commits, we can reuse a physical register.

Floating Point Number

Exponent	significant	object
0	0	0
0	nonzero	Denorm
1-254	anything	+/- floating point #.
255	0	+/-∞
255	nonzero	NAN

Denorm: No leading 1. implicit expo -126. ∴ smallest pos #: $2^{-23} \times 2^{-126}$

Cache

Tag	Set Index	Block offset

Size of Block offset = $\log_2$ # of bytes / block
Size of index: $\log_2$(number of sets) = $\log_2(\frac{\#\text{ of lines}}{N})$
Size of Tag: Address size − 上面两个
Valid Bit: indicate whether this tag is valid for this program
Cache Flush: invalidate all entries
Victim Cache: a small fully associative cache
Cache Philosophy: programmer-invisible hardware mechanism to give illusion of speed of fastest memory with size of largest memory.

Use of Principle of Locality: program access small portion of address space at any instant of time (spatial) and repeatedly access that portion (temporal)

Cache write:
Write Back: write only to cache and then write cache block back to memory only when evicted.
array[i] = array[j] + j ① read ② write 准备 ③ write back
Write Through: write cache + write through to memory. If store miss, first load the line
Write allocate:
No write allocate: Just write to the memory.

Cache Replacement Policy: second order effect: only happen on misses
① Random
② Least Recently used — small sets
③ First In First Out — highly associative (Round Robin)
④ Not Most Recently used — (FIFO with exception of the most recently used)

Direct Mapped cache: N = 1 = associativity
Line goes 1 place; only need 1 comparator. 比 tag
If multiple words in a block, then also need a mux to select which word's data to return.
Fully associated: N = number of lines
Line can go anywhere; 1 sets; ∴ No index field
N-way Set Associative: N places for a line
N comparator. N×1 select mux for comparator to check hit or not.

Total cache capacity = Associativity (N) × # of sets (S) × blocksize (B) C = NSB

Average Memory Access Time (AMAT) = Hit time + Miss Rate × Miss Panelty
Hit time: cache 7 hit time
Local miss rate $L2 = \frac{L2\text{ misses}}{L1\text{ misses}}$ } 大于/等于
Global miss rate $L2 = \frac{L2\text{ misses}}{\text{Total access}}$ — cr-level
= Local $L1 × local $L2

AMAT = L1 hit + L1 miss × (L1 hit + L2 miss × (L3 hit + L3 miss × L3 miss panelty)

3C: total defines miss rate.
1. Compulsory (cold start / process migration)

Solution: block size ++. (increase miss penalty, very large increase miss rate).
Capacity:
solution: cache size ++ (may increase access time)
Conflict:
solution: ① cache size ++
② associativity ++ (may increase access time.
Bad for conflict: block size ++.

Performance = 1/Execution time
{ Latency (execution time)
{ Bandwidth (throughput) Task completed per unit of time
Iron Law of performance:
CPU $\frac{\text{Time}}{\text{Program}} = \frac{\text{Clock cycles}}{\text{Program}} \times$ Clock cycle time
$= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Average clock cycles}}{\text{Instruction}}$ (CPI) × clock cycle time = $\frac{1}{\text{Clock Rate}}$
instruction count ←

WorkLoad & Benchmark:
workload: set of programs run on a computer
benchmark: (standardized) programs selected for use in comparing performance
SPEC: (system performance evaluation cooperative)
{ 12 integer programs
{ 17 FP programs
SPEC ratio = $\frac{\text{old reference execution time}}{\text{new execution time}}$
$\sqrt[n]{\prod_{i=1}^{n} \text{Execution time ratio}_i}$: Give same relative answer no matter what computer is used as reference

Flynn Taxonomy
Single Instruction / Single Data Stream SISD
e.g. superscalar, RISV processor
Single Instruction / Multiple Data Stream SIMD
← hardware support needed
Multiple Instruction / Multiple Data Stream MIMD
e.g. Multicore, warehouse-scale computers
Multiple Instruction / single data stream MISD
(SPMD: simple program multiple data)
Single program runs on all processors of a MIMD, use synchronization primitive to do cross processor execution coordination

Amdahl's Law
Speedup = $\frac{1}{1-F+\frac{F}{S}}$ — F: fraction < 1, S: factor > 1, speed up
Strong scaling: 问题规模不变, on a parallel processor
Weak scaling: 问题规模随 processors # 等比例增加.
Load Balance: 每个 processor work → (same)

Intrinsics: C functions and procedures for inserting assembly language into C code.

Parallelism: the only path to higher performance
① Sequential processor performance not expected to increase much, might even ↓. ② Apps with more capability ↓. clock rate & CPI flat.
③ In mobile systems, use multiple cores & GPU.
④ WSC, use multiple nodes, and all the MIMD / SIMD capability of each node.

Thread: a sequence flow of insts that perform some tasks.

OS Thread: OS Time-multiplexing software threads onto hardware threads.
- Active a software thread: load its regs and PC from memory into a hardware thread's regs and jump to its saved PC.
- Disactive: interrupt, save its PC & regs.

Hardware Multithreading: utilize expensive processor resource (Hyper Threading)

OPENMP #include <omp.h>

compile: gcc prog.c -fopenmp

Programming model: Fork-join

1. 变量属性: 本是 shared.
   ① #pragma omp parallel private(x) → make private.
   ② #pragma omp parallel for → make loop index private (默认行为)
2. openmp thread: OS thread.
   omp_set_num_threads(x) → set manually
   a = omp_get_num_threads() → # of thread
   a = omp_get_thread_num() → ID
3. Data race: Different threads try to access a same location in memory, and at least 1 is write.

RISC-V 2 solutions

<-> Read-write Pair

lr rd, rs: Load the word pointed to by rs into rd, and add a reservation.

sc rd, rs1, rs2: Store rs2 to MEM(rs1) only if the reservation of rs1 is valid and set status in rd. Return 0 when rs1 不变 since lr; Return nonzero (failure) when rs1 变了, 实际 store 不会进行.

```
try:
    lr  t1, s1       # load reserved
    sc  t0, s1, s4   # store conditional
    bne t0, x0, try  If another thread runs sc, this will fail.
    add s4, x0, t1
```

Test-and-set (1-set; 0-not)

Test to see if a memory location is set; set it to 1 if it is not, otherwise keep testing.

```
try:                    Locked:
    lr t1, s1             # critical section
    bne t1, x0, try     unlock:
    sc t0, s1, t2         sw x0, 0(s1)
    bne t0, x0, try
```

<=> RISV Atomic Memory Operations (AMO)

Atomatically perform an operation on an operand in memory and set the destination register to the original memory value.

e.g. amoadd.w rd, rs2, (rs1):
- t = M[x[rs1]]
- x[rd] = t
- M[x[rs1]] = t + x[rs2]

4. for ③这行 for 内部的元素都是 private.
   ① Divide index region sequentially $0-\frac{n}{2} \mid \frac{n}{2}-n$
   ② No premature exits from the loop allowed (break, return, exit, goto) ← Interrupt 可能出现.
5. Timing double omp_get_wtime(void)
6. OpenMP reduction (解决 data race)
   #pragma omp for reduction(运算符: 变量名)
   Var is private to each thread.

Memory Mapped I/O:

- 部分 address 对应于 I/O 设备中的 registers

<-> Polling: processor reads control register of device in loop until device sets Ready Bit in control reg to 1. Then processor load / write. e.g. Mouse

<=> Interrupt: device interrupts program when I/O is ready, return when done with data transfer.

interrupt handler: functions called when an interrupt exists. CPU 中有一个 interrupt table 存了 address.

jal to handler & store any state. Handler run on current stack and returns on finish (Thread 不会注意到)

- Interrupt: external (e.g. key press) Asynchronous 不会造成 该回不再有次
- Exception: internal (e.g. page fault, bus error, illegal insts). Synchronous. 当句解决 must.
- Trap: action of servicing interrupt or exception by hardware jump to "trap handler code"

Supervisor mode: enforce resource constraints to apps.

Process enters the supervisor mode by using an interrupt, and change out of it using a special inst.

Syscall: call into an OS routine

set up function arguments in registers, and then raise software interrupt. Then OS does it work and return to user mode.

⇒ OS mediate access to all resources, including devices & CPU timing

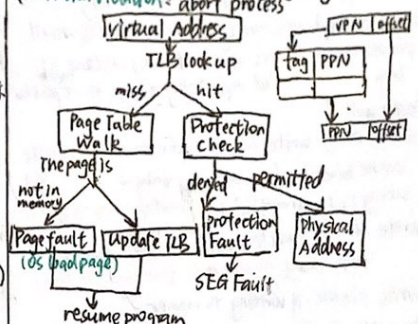
Multiprogramming

~~context switch~~: when jumping into process, set timer interrupt; when it expires, store PC, registers, etc; pick a different process to run and load its state, set timer, change to user mode, jump to new PC.

Scheduling: 选择下一个 process, decide

Virtual Memory

1. Base and Bound registers are visible/accessible only when processor is running in supervisor mode
2. Hierarchical Page Table exploits sparsity of virtual address space use
3. Every inst and data access needs address translation and protection checks.
4. Translation Lookaside Buffer: cache some Translation
   - TLB miss = hardware / software mechanism to refill TLB
   - Page fault = precise trap. So handler can easily resume after retrieving page.
   - Protection violation = abort process



5. TLB reach: size of largest virtual address space that can be simultaneously mapped by TLB.
6. Virtual Machine Mode: host OS activates virtual execution mode for guest OS; Guest OS think it runs in supervisor mode, but in fact it doesn't have access to physical memory.
   Guest virtual → Guest physical → Machine page.
7. Cloud server: virtualization + distributed storage
8. Prefetch accuracy = $\frac{\text{used prefetches}}{\text{sent prefetches}}$

Embedded system 特点

① Specifically-functioned
② Tightly constrained: low cost, power, fast, single- → to fewer components based
③ Relative and real time: must continually monitor the desired environment and react to change
④ Hardware and software co-existence

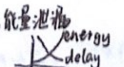
Embedded system 例子

① communication devices: wired and wireless routers and switches
② Automotive apps: breaking system, traction control, airbag release system, and cruise-control apps
③ Aerospace apps: flight-control systems, engine controller, auto-pilots and passenger in-flight entertainment syss.
④ Defense systems: radar systems, fighter aircraft flight-control systems, radio systems, and missle guidance system.

Embedded system design matrics

① Timing performance
② Power consumption: 动态耗能, 短路, 能量泄漏

$P_{dynamic} = \alpha \cdot C_{phy} \cdot V_{dd}^2 \cdot f_{clk}$ (α: switching activity 电路)



ASIC Application-Specific Integrated Circuit

	Processor	FPGA	ASIC
性:	可编程	可配置	不可配置 (configurable ×)
order:	sequential	parallel wired	no wire
power:	多		
	Transfer bottleneck	distributed data	

Dynamically-Linked Library & static

DLL advantages: ① program requires less disk space if DLL is shared between programs ② execution of 2 programs requires less memory, also slight performance gain due to better caching ③ a replaced library 改进 improves the program that use this library

DLL disadvantage: ① At loading or runtime there is overhead to dynamically link ② Having the executable is not enough, we still need the correct major version of the library.

Single cycle CPU inefficient:

Not all insts exercise the critical path. It is not parallelized. Each component can be active concurrently. Slow.

What does OS do?

① One of the 1st things that runs when your computer starts (right after firmware/bootloader)
② Load, run and manage programs:
   a. Multiple programs at the same time (time sharing)
   b. Isolate programs from each other (isolation)
   c. multiplex resources between apps e.g. apps
③ Services: File system, Network stack, printer, etc.
④ Find and control all the devices in the machine in a general way (using "device driver")

What does OS core do?

① Provide interaction with outside world (with devices)
② Provide isolation between running programs (processes)

开机过程

① BIOS: find storage device (HDD, flash) and load first sector (bootloader)
② Bootloader: (stored on disk) load OS kernel from file system of the HDD into memory and jump into it
③ OS Boot: initialize services, drivers, etc.
④ Init: Launch an app that waits for input in loop.

- Basic Input/Output System
- Unified Extensible Firmware Interface: successor of BIOS

Handling Trap in Inorder Pipeline

① Hold exception flags in pipeline until commit point
② Exceptions in earlier inst / pipestage override exceptions in later inst / pipe stage
③ Inject external interrupts at commit point
④ If exception/interrupt at commit: update Cause and SEPC registers, kill all stages, inject handler PC into fetch stage.

Launching Application:

① Created by another process calling into an OS routine using syscall
② Loads exe file from disk using the file system service and puts instrs & data into memory, prepare stack & heap ③ set argc and argv, jump into main function.

3 Reasons for Virtual Memory:

① Add Disks to Hierarchy
② Simplify memory for apps (give them a virtual view of memory)
③ Protection between processes (OS controls translation mechanism between virtual and physical address.

Field Programmable Gate Array

三个组成:
- Functional Block
- I/O Block
- Routing network

} all can be reprogrammed with configurations to implement app-specific digital circuit.

Configuration File: ← 存在 a bit stream.

1. Functional Block = Look Up Table type.
   consists of different # of LEs. LE 有 different size.
   Input: n output: 1. ⇒ represent 2^{2^n} function
   其他计算资源: carry logic, embedded memory blocks, multiplier and other hard cores.
2. Routing
   Switch block at ✳

Hardware Description Language: VHDL & verilog.

Warehouse Scale Computers 特性.

1. ample Parrallelism
   - Request-level parallelism e.g. web search
   - Data-level parallelism e.g. image classifier Training
2. Scale
   - Scale of economy: low per-unit cost
   - High number of risk: disk fail / hour
3. Operation cost count
   - long-lifetime
   - cost of equipment purchases ≪ cost of ownership

server → rack → ... Array (cluster)

WSC Storage Hierarchy:

Lower latency to DRAM in another server than local disk; Higher bandwidth to local disk than to DRAM in another server.

WSC Power Uses:

Power Usage Effectiveness = Total Building Power / IT Equipment Power

为 server/networking 的 efficiency 考虑

理想: consume no power when idle; gradually consume more power as the activity level ↑.

Request-Level Parallelism RLP

e.g. Websearch

1. Implementation Strategy:
   ① randomly distribute entries
   ② make many copies of search data (cat)
   ③ load balance requests across replicas.
2. Redundant copies of indices and documents.

好处: ① Breakup hot spots
② increase opportunity for request-level parallelism
③ make system more tolerant of failures.

Data Level Parallelism DLP

SIMD: 单一机器, 数对指令 & 硬件

DLP on WSCs: 多台机器, map reduce & scalable filesystem

Map Reduce:

1. 隐藏 underlying runtime system:
   ① Automatically parallelize the computation across large scale clusters of machines
   ② Handle machine failure
   ③ schedule inter-machine communication to make efficient use of the networks
2. 流行

① Map(inputkey: 文件名, Inputvalue) → list(中间key, 中间val)
e.g. map(key, value) key: doc name, value: doc content
for each word w in value:
emit(w, 1).

② Reduce(中间 key, list of 中间 val) → list(output-value)
e.g. reduce(key, values) key: word, value: list(counts)
result = 0
for each v in values: (word, count sum)
result += v
emit(key, result)

抽象化执行:
① split inputs, startup programs on a cluster of machines
② Assign map & reduce to idle workers
③ Perform a map task, generate intermediate key/value pairs.
④ write to the buffers
⑤ read intermediate key/value pairs, sort them by key.
⑥ Perform a reduce task for each intermediate key, write the result to the output-files.

Map Reduce

Coin Pair:
String person
String coinType

1. 计算 number of coins of each denomination that a person has
2. 计算有多少钱 用一个 value of coin(String coinType)

1. ① map(Coinpair pair):
   emit(pair, 1)
   ② reduce(Coinpair pair, Iterable<int> count):
   total = 0
   for num in count:
   total += num
   emit(pair, total)

2. ① map(tuple<Coinpair, int> output):
   pair, amount = output
   emit(pair.person, value of Coin(pair.coinType).amount)
   ② Reduce(String person, Iterable<float> values):
   total = 0
   for i in values:
   total += i
   emit(person, total).

Spark & Hadoop

Hadoop: Opensource MapReduce Framework, has a Hadoop Distributed File System

Spark: high abstraction of map reduce

Resilient Distributed Datasets (RDD) are the primary abstraction of a distributed collection of items.

- Transform: RDD → RDD
  - map(f): return a new transformed item formed by calling f on a source element
  - ② flatmap(f): return a sequence rather than a single item, each input item can be mapped to 0 or more output items.
  - ③ reduceByKey(f): when called on a dataset of (K,V) pairs, returns a dataset of (K,V) pairs where values for each key are aggregated using f: (V,V) → V.
- Actions RDD → value
  - reduce(f). 不管 key. 直接 aggregate to 结果.
  - sc.parallelize(data): parallelize a python collection, data.

Example:

coinData = sc.parallelize(coinPairs). → (person, coinpair)

out1 = coinData.map(lambda(k1,k2): ((k1,k2),1)).reduceByKey(lambda v1,v2: v1+v2)

out2 = out1.map(lambda (k,v): ([k[0], v·value of coin(k[1])])).reduceByKey(lambda v1,v2: v1+v2).

wordcount example:

text_RDD.flatmap(lambda x: x.split(' '))
.map(lambda x: (x,1))
.reduceByKey(lambda x,y: x+y).

Hamming ECC

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
	p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8	d9	d10	d11	p16	d12
p1	X		X		X		X		X		X		X		X		X
p2		X	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		
p16																X	X

纵向是偶数.

Hamming distance = different in # of bits

Min Hamming distance of valid parity code is 2.

简单的 1 parity 又能 detect odd number of errors.

Hamming code 可以检测 2 bits error, 纠正 1 bit error

Hamming code 不可 detect all 3 bit errors.

Dependability via redundancy

- Spatial redundancy: replicated data or check information or hardwares to handle hard & soft failures
- Temporal redundancy: redundancy in time to handle soft failures

Dependability Measures:

Dependability: Mean Time To Failure (MTTF)

Service interruption: Mean Time To Repair (MTTR)

MTBF: mean time between failure

MTBF = MTTF + MTTR

Availability = MTTF ↑ / (MTTF + MTTR) ↓

Number of 9s: 3: 99.9%

Reliability Measures:

AFR: annualized failure rate

e.g. 1000 disks, 10000 hours MTTF

1000 disk × 24 hs × 365 days / 10000 hours

= 87.6 failed. ⇒ AFR = 8.76%

Dependability Design Principle:

Design principle: no single point of failure

为 Amdahl's law 服务

- soft error: 自己恢复
- hard error: disk

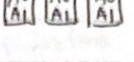
EDC / ECC

RAID: redundant arrays of inexpensive disks

Direct Memory Access DMA

Allow I/O devices to directly read/write main memory; Hardware supported by the DMA Engine.

1. DMA Engine
   Contains registers written by CPU:
   Memory address to place data; # of bytes; I/O device #, direction of transfer; unit of transfer, amount to transfer per burst.
2. DMA: Incoming data
   ① Receive interrupt from device
   ② CPU takes interrupt, begins transfer
   - instruct DMA engine/device to place data at certain address
   ③ Device / DMA engine handle the transfer
   - CPU free to do other things
   ④ Upon completion, device / DMA Engine interrupt the CPU again
3. DMA: Outgoing data
   ① CPU decides to initiate transfer, confirms that external device is ready
   ② CPU begins transfer
   - instruct DMA engine / device that data is available at certain address
   ③ Device / DMA engine handle the transfer
   - CPU free to do other things
   ④ Device / DMA engine interrupt the CPU again to signal completion.
4. Arbitrate between CPU and DMA Engine / Device access to memory.
   ① Burst mode
   start transfer of data block, CPU cannot access memory in the meantime
   ② Cycle stealing Mode
   DMA engine transfer a byte, release control, then repeats - interleaves processor / DMA engine access.
   ③ Transparent Mode
   DMA transfer only occurs when CPU is not using the system bus.

	Configuration	Pro / Good for	Con / Bad for	picture
RAID 0 (striping)	split data across multiple disks	No overhead, fast read & write	Reliability	
RAID 1	Mirrored disks: extra copy of data	Fast read/write, fast recovery	High overhead (capacity)	
RAID 3	Byte level stripping with single parity disk	smallest overhead to check parity	Need to read all disks, even for read, to detect errors	
RAID 4	Block level stripping with single parity disk	Higher throughput for small reads	Still slow for small writes	
RAID 5	Block level stripping, parity distributed across disks	Higher throuput of small writes	the time to repair a disk is so long that another disk might fail at the same time	

Meltdown: out of order; 目标 leak/dump memory

We want to read from OS pages that hold data of other processes. Those are in our VirtualMem space in order for the OS to process them quickly, but it is forbidden for our process to read them. If we try to read them, the OS will check (with an if) if we are allowed to read them and thus return with a page-fault. But speculative execution will nevertheless cause those pages to be read, but the process doesn't see the result. We use this to take the to-be-read value and use it as an offset in an array. We later check all the array if it is in the cache (using timing) - if so we know which range of value the variable that we were not supposed to read had.

```
raise exception();
access(probe-array[secret * 4096])
```

Disk

Disk access time = seek time + rotation time + transfer time + controller overhead

rotation time: $\frac{1}{2}$ · one cycle (s).

seek time: $\frac{1}{3}$ · number of tracks × time to move across one track

	Definition	Pro / Good for	Con
Polling	Forces the hardware to wait on ready bit. (Alternatively, if timing of device is known, the ready bit can be polled at the frequency of the device)	Low Latency; low overhead when data is available. Good for devices that are always busy or when you can't make progress until the device replies	Can't do anything else while polling, Can't sleep while polling (CPU 全速运行)
Interrupts.	Hardware fires an "exception" when it becomes ready. CPU changes PC register to execute code in the interrupt handler when this occurs	Can do useful work while waiting for response; can wait on many things at once. Good for: devices that take a long time to response, especially if you can do other work while waiting	Nondeterministic when interrupt occurs; interrupt handler has some overhead (e.g. save all registers, flush pipeline, etc). Higher latency per event. Worse throughput.

Network

1. Shared vs Switch:
   - Shared: data time pairs communicate at same time
   - Aggregate bandwidth (BW) in switched network is many times that of shared
2. Links connect switches and/or routers to each other and to computers / devices
3. Layering, redundancy, protocals, encapsulation → abstraction
4. Hierarchy of layers
   - App (chat client, game, ...)
   - Transport (TCP, UDP)
   - Network (IP)
   - Data link layer (Ethernet)
   - Physical link (copper, wireless, etc)
5. Protocal: packet structure and control commands to manage communication

   Protocal family: a set of cooperating protocals that implement the network stack.

TCP/IP: Transmission Control Protocal / Internet Protocol.

Spectre: branch prediction

目标: read out a victim's sensitive info

```
if (x < array1_size)        <- cache miss, so run next line due to prediction history
    y = array2[array1[x] * 4096]
```

array1[x] cache hit, as k is cached, so load array2[k·4096]

Prerequisite:

1. array1[x], with an out of bound x (larger than array1_size, resolves to a secrete byte k that is cached.
2. array1_size and array2 uncached
3. previous x value have been valid.

Regarding a misprediction with an illegal x, array2[k·4096] will not be used, but k has been loaded into CPU cache. We can use Flush+Reload to guess k with array2.

Flush+Reload

① Permission check for cflush

② disallowance of memory sharing

③ Tuning software programs

PSEUDO INSTRUCTIONS ③

MNEMONIC	NAME	DESCRIPTION	USES
beqz	Branch = zero	if(R[rs1]==0) PC=PC+{imm,1b'0}	beq
bnez	Branch ≠ zero	if(R[rs1]!=0) PC=PC+{imm,1b'0}	bne
fabs.s,fabs.d	Absolute Value	F[rd] = (F[rs1]< 0) ? –F[rs1] : F[rs1]	fsgnx
fmv.s,fmv.d	FP Move	F[rd] = F[rs1]	fsgnj
fneg.s,fneg.d	FP negate	F[rd] = –F[rs1]	fsgnjn
j	Jump	PC = {imm,1b'0}	jal
jr	Jump register	PC = R[rs1]	jalr
la	Load address	R[rd] = address	auipc
li	Load imm	R[rd] = imm	addi
mv	Move	R[rd] = R[rs1]	addi
neg	Negate	R[rd] = –R[rs1]	sub
nop	No operation	R[0] = R[0]	addi
not	Not	R[rd] = ~R[rs1]	xori
ret	Return	PC = R[1]	jalr
seqz	Set = zero	R[rd] = (R[rs1]== 0) ? 1 : 0	sltiu
snez	Set ≠ zero	R[rd] = (R[rs1]!= 0) ? 1 : 0	sltu

ARITHMETIC CORE INSTRUCTION SET ②

RV64M Multiply Extension

MNEMONIC	FMT	NAME	DESCRIPTION (in Verilog)	NOTE
mul,mulw	R	MULtiply (Word)	R[rd] = (R[rs1] * R[rs2])(63:0)	1)
mulh	R	MULtiply High	R[rd] = (R[rs1] * R[rs2])(127:64)	
mulhu	R	MULtiply High Unsigned	R[rd] = (R[rs1] * R[rs2])(127:64)	2)
mulhsu	R	MULtiply upper Half Sign/Uns	R[rd] = (R[rs1] * R[rs2])(127:64)	6)
div,divw	R	DIVide (Word)	R[rd] = (R[rs1] / R[rs2])	1)
divu	R	DIVide Unsigned	R[rd] = (R[rs1] / R[rs2])	2)
rem,remw	R	REMainder (Word)	R[rd] = (R[rs1] % R[rs2])	1)
remu,remuw	R	REMainder Unsigned (Word)	R[rd] = (R[rs1] % R[rs2])	1,2)

RV64A Atomtic Extension

amoadd.w,amoadd.d	R	ADD	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] + R[rs2]	9)
amoand.w,amoand.d	R	AND	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] & R[rs2]	9)
amomax.w,amomax.d	R	MAXimum	R[rd] = M[R[rs1]], if (R[rs2] > M[R[rs1]]) M[R[rs1]] = R[rs2]	9)
amomaxu.w,amomaxu.d	R	MAXimum Unsigned	R[rd] = M[R[rs1]], if (R[rs2] > M[R[rs1]]) M[R[rs1]] = R[rs2]	2,9)
amomin.w,amomin.d	R	MINimum	R[rd] = M[R[rs1]], if (R[rs2] < M[R[rs1]]) M[R[rs1]] = R[rs2]	9)
amominu.w,amominu.d	R	MINimum Unsigned	R[rd] = M[R[rs1]], if (R[rs2] < M[R[rs1]]) M[R[rs1]] = R[rs2]	2,9)
amoor.w,amoor.d	R	OR	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] | R[rs2]	9)
amoswap.w,amoswap.d	R	SWAP	R[rd] = M[R[rs1]], M[R[rs1]] = R[rs2]	9)
amoxor.w,amoxor.d	R	XOR	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] ^ R[rs2]	9)
lr.w,lr.d	R	Load Reserved	R[rd] = M[R[rs1]], reservation on M[R[rs1]]	
sc.w,sc.d	R	Store Conditional	if reserved, M[R[rs1]] = R[rs2], R[rd] = 0; else R[rd] = 1	

CORE INSTRUCTION FORMATS

	31 27	26 25	24 20	19 15	14 12	11 7	6 0
R	funct7		rs2	rs1	funct3	rd	Opcode
I	imm[11:0]			rs1	funct3	rd	Opcode
S	imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode
SB	imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]					rd	opcode
UJ	imm[20|10:1|11|19:12]					rd	opcode

REGISTER NAME, USE, CALLING CONVENTION ④

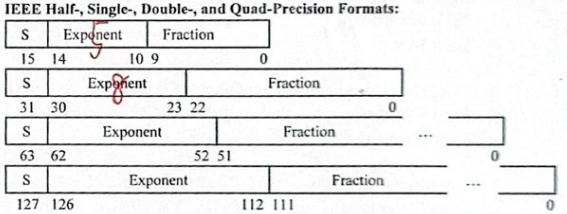
REGISTER	NAME	USE	SAVER
x0	zero	The constant value 0	N.A.
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	--
x4	tp	Thread pointer	--
x5-x7	t0-t2	Temporaries	Caller
x8	s0/fp	Saved register/Frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-a1	Function arguments/Return values	Caller
x12-x17	a2-a7	Function arguments	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporaries	Caller
f0-f7	ft0-ft7	FP Temporaries	Caller
f8-f9	fs0-fs1	FP Saved registers	Callee
f10-f11	fa0-fa1	FP Function arguments/Return values	Caller
f12-f17	fa2-fa7	FP Function arguments	Caller
f18-f27	fa2-fs11	FP Saved registers	Callee
f28-f31	ft8-ft11	R[rd] = R[rs1] + R[rs2]	Caller

IEEE 754 FLOATING-POINT STANDARD

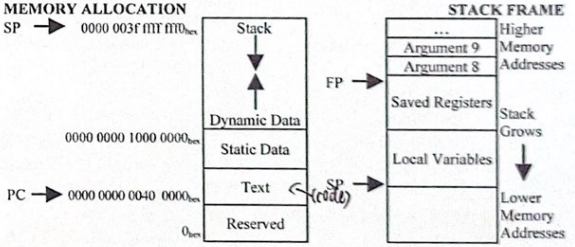
$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$

where Half-Precision Bias = 15, Single-Precision Bias = 127,
Double-Precision Bias = 1023, Quad-Precision Bias = 16383

IEEE Half-, Single-, Double-, and Quad-Precision Formats:



MEMORY ALLOCATION



STACK FRAME

SIZE PREFIXES AND SYMBOLS

SIZE	PREFIX	SYMBOL	SIZE	PREFIX	SYMBOL
10^3	Kilo-	K	2^{10}	Kibi-	Ki
10^6	Mega-	M	2^{20}	Mebi-	Mi
10^9	Giga-	G	2^{30}	Gibi-	Gi
10^{12}	Tera-	T	2^{40}	Tebi-	Ti
10^{15}	Peta-	P	2^{50}	Pebi-	Pi
10^{18}	Exa-	E	2^{60}	Exbi-	Ei
10^{21}	Zetta-	Z	2^{70}	Zebi-	Zi
10^{24}	Yotta-	Y	2^{80}	Yobi-	Yi
10^{-3}	milli-	m	10^{-15}	femto-	f
10^{-6}	micro-	μ	10^{-18}	atto-	a
10^{-9}	nano-	n	10^{-21}	zepto-	z
10^{-12}	pico-	p	10^{-24}	yocto-	y

RV64I BASE INTEGER INSTRUCTIONS, in alphabetical order

MNEMONIC	FMT	NAME	DESCRIPTION (in Verilog)	NOTE
add, addw	R	ADD (Word)	R[rd] = R[rs1] + R[rs2]	1)
addi, addiw	I	ADD Immediate (Word)	R[rd] = R[rs1] + imm	1)
and	R	AND	R[rd] = R[rs1] & R[rs2]	
andi	I	AND Immediate	R[rd] = R[rs1] & imm	
auipc	U	Add Upper Immediate to PC	R[rd] = PC + {imm, 12'b0}	
beq	SB	Branch EQual	if(R[rs1]==R[rs2) PC=PC+{imm,1b'0}	
bge	SB	Branch Greater than or Equal	if(R[rs1]>=R[rs2) PC=PC+{imm,1b'0}	
bgeu	SB	Branch ≥ Unsigned	if(R[rs1]>=R[rs2) PC=PC+{imm,1b'0}	2)
blt	SB	Branch Less Than	if(R[rs1]<R[rs2) PC=PC+{imm,1b'0}	
bltu	SB	Branch Less Than Unsigned	if(R[rs1]<R[rs2) PC=PC+{imm,1b'0}	2)
bne	SB	Branch Not Equal	if(R[rs1]!=R[rs2) PC=PC+{imm,1b'0}	
ebreak	I	Environment BREAK	Transfer control to debugger	
ecall	I	Environment CALL	Transfer control to operating system	
jal	UJ	Jump & Link	R[rd] = PC+4; PC = PC + {imm,1b'0}	
jalr	I	Jump & Link Register	R[rd] = PC+4; PC = R[rs1]+imm	3)
lb	I	Load Byte	R[rd] = {56'bM[](7),M[R[rs1]+imm](7:0)}	4)
lbu	I	Load Byte Unsigned	R[rd] = {56'b0,M[R[rs1]+imm](7:0)}	
ld	I	Load Doubleword	R[rd] = M[R[rs1]+imm](63:0)	
lh	I	Load Halfword	R[rd] = {48'bM[](15),M[R[rs1]+imm](15:0)}	4)
lhu	I	Load Halfword Unsigned	R[rd] = {48'b0,M[R[rs1]+imm](15:0)}	
lui	U	Load Upper Immediate	R[rd] = {32b'imm<31>, imm, 12'b0}	
lw	I	Load Word	R[rd] = {32'bM[](31),M[R[rs1]+imm](31:0)}	4)
lwu	I	Load Word Unsigned	R[rd] = {32'b0,M[R[rs1]+imm](31:0)}	
or	R	OR	R[rd] = R[rs1] | R[rs2]	
ori	I	OR Immediate	R[rd] = R[rs1] | imm	
sb	S	Store Byte	M[R[rs1]+imm](7:0) = R[rs2](7:0)	
sd	S	Store Doubleword	M[R[rs1]+imm](63:0) = R[rs2](63:0)	
sh	S	Store Halfword	M[R[rs1]+imm](15:0) = R[rs2](15:0)	
sll, sllw	R	Shift Left (Word)	R[rd] = R[rs1] << R[rs2]	1)
slli, slliw	I	Shift Left Immediate (Word)	R[rd] = R[rs1] << imm	1)
slt	R	Set Less Than	R[rd] = (R[rs1] < R[rs2]) ? 1 : 0	
slti	I	Set Less Than Immediate	R[rd] = (R[rs1] < imm) ? 1 : 0	
sltiu	I	Set < Immediate Unsigned	R[rd] = (R[rs1] < imm) ? 1 : 0	2)
sltu	R	Set Less Than Unsigned	R[rd] = (R[rs1] < R[rs2]) ? 1 : 0	2)
sra, sraw	R	Shift Right Arithmetic (Word)	R[rd] = R[rs1] >> R[rs2]	1,5)
srai, sraiw	I	Shift Right Arith Imm (Word)	R[rd] = R[rs1] >> imm	1,5)
srl, srlw	R	Shift Right (Word)	R[rd] = R[rs1] >> R[rs2]	1)
srli, srliw	I	Shift Right Immediate (Word)	R[rd] = R[rs1] >> imm	1)
sub, subw	R	SUBtract (Word)	R[rd] = R[rs1] – R[rs2]	1)
sw	S	Store Word	M[R[rs1]+imm](31:0) = R[rs2](31:0)	
xor	R	XOR	R[rd] = R[rs1] ^ R[rs2]	
xori	I	XOR Immediate	R[rd] = R[rs1] ^ imm	

OPCODES IN NUMERICAL ORDER BY OPCODE

MNEMONIC	FMT	OPCODE	FUNCT3	FUNCT7 OR IMM	HEXADECIMAL
lb	I	0000011	000		03/0
lh	I	0000011	001		03/1
lw	I	0000011	010		03/2
ld	I	0000011	011		03/3
lbu	I	0000011	100		03/4
lhu	I	0000011	101		03/5
lwu	I	0000011	110		03/6
addi	I	0010011	000		13/0
slli	I	0010011	001	0000000	13/1/00
slti	I	0010011	010		13/2
sltiu	I	0010011	011		13/3
xori	I	0010011	100		13/4
srli	I	0010011	101	0000000	13/5/00
srai	I	0010011	101	0100000	13/5/20
ori	I	0010011	110		13/6
andi	I	0010011	111		13/7
auipc	U	0010111			17
addiw	I	0011011	000		1B/0
slliw	I	0011011	001	0000000	1B/1/00
srliw	I	0011011	101	0000000	1B/5/00
sraiw	I	0011011	101	0100000	1B/5/20
sb	S	0100011	000		23/0
sh	S	0100011	001		23/1
sw	S	0100011	010		23/2
sd	S	0100011	011		23/3
add	R	0110011	000	0000000	33/0/00
sub	R	0110011	000	0100000	33/0/20
sll	R	0110011	001	0000000	33/1/00
slt	R	0110011	010	0000000	33/2/00
sltu	R	0110011	011	0000000	33/3/00
xor	R	0110011	100	0000000	33/4/00
srl	R	0110011	101	0000000	33/5/00
sra	R	0110011	101	0100000	33/5/20
or	R	0110011	110	0000000	33/6/00
and	R	0110011	111	0000000	33/7/00
lui	U	0110111			37
addw	R	0111011	000	0000000	3B/0/00
subw	R	0111011	000	0100000	3B/0/20
sllw	R	0111011	001	0000000	3B/1/00
srlw	R	0111011	101	0000000	3B/5/00
sraw	R	0111011	101	0100000	3B/5/20
beq	SB	1100011	000		63/0
bne	SB	1100011	001		63/1
blt	SB	1100011	100		63/4
bge	SB	1100011	101		63/5
bltu	SB	1100011	110		63/6
bgeu	SB	1100011	111		63/7
jalr	I	1100111	000		67/0
jal	UJ	1101111			6F
ecall	I	1110011	000	000000000000	73/0/000
ebreak	I	1110011	000	000000000001	73/0/001

Notes:
1) *The Word version only operates on the rightmost 32 bits of a 64-bit registers*
2) *Operation assumes unsigned integers (instead of 2's complement)*
3) *The least significant bit of the branch address in jalr is set to 0*
4) *(signed) Load instructions extend the sign bit of data to fill the 64-bit register*
5) *Replicates the sign bit to fill in the leftmost bits of the result during right shift*
6) *Multiply with one operand signed and one unsigned*
7) *The Single version does a single-precision operation using the rightmost 32 bits of a 64-bit F register*
8) *Classify writes a 10-bit mask to show which properties are true (e.g., –inf, -0, +0, +inf, denorm, ...)*
9) *Atomic memory operation; nothing else can interpose itself between the read and the write of the memory location*

The immediate field is sign-extended in RISC-V