Ch.1 Computer Abstraction and Technology

1. Abstraction:
    - Hiding lower-level details when they aren't important
    - Providing only required interface information b/t adjacent layers
    - Helps us deal with complexity
2.
- Application software: written in HLL
- System software:
    - HLL → Compiler → Assembly language program → Assembler → Machine Code
    - HLL: level of abstraction closer to problem domain;
    - Assembly language: textual representation of instructions
    - OS: service code manages computer hardware
        - handling input/ output
        - managing memory and storage
        - scheduling tacks & sharing resources
- Hardware
    - Processor, memory, I/O controllers

Instruction set architecture(ISA): the hardware / software interface

3. Performance
- Performance = 1/Execution Time
- X is n time faster than Y
    - Performance x/ Performance y = Execution time y / Execution time x
- Measuring performance:
    - Elapsed time/ response time 运行程序期间：total response time including processing, I/O, OS overhead, idle
        - referred by system performance
    - CPU (execution) time: the actual time CPU spends computing for a specific task
        - user CPU time (spent in a program itself) + system CPU time(spent in OS)
        - referred by CPU performance

4. CPU clocking
    - clock period: duration of a clock cycle
        - 250ps = 0.25ns = 250*10^(-12) s
    - clock frequency(rate): cycles per second
        - 4.0GHz = 4000MHz = 4.0*10^9 Hz
    - CPU time = CPU clock cycles * Clock cycle time
      = CPU clock cycles / Clock rate
      = Instruction n count * CPI * Clock cycle time
      = Instruction n count * CPI / Clock rate
      $= \frac{Instructions}{Program} * \frac{clock\ cycles}{Instruction} * \frac{Seconds}{ClockCycles}$

- Instruction Count for a program: determined by program, ISA, compiler(kinds / options)
- Average CPI: determined by CPU hardware architecture
    - i7 designed for high-performance, aeron designed for low power consumption (don't care about CPI)
    - **Even same compiler(#IC same) + clock frequency same, performance can be different**
- Performance summary
    - depends on:
        - Algorithm: affects IC, possibly CPI
        - Programming language: affects IC, CPI
        - Compiler: affects IC, CPI
        - ISA: affects IC, CPI, Tc

- MIPS: Millions of Instructions Per Second
    - $$\text{MIPS} = \frac{Instruction\ count}{Execution\ time * 10^6}$$

$$= \frac{Instruction\ count}{Instruction\ count * \frac{CPI}{clock\ rate} * 10^6}$$

$$= \frac{clock\ rate}{CPI * 10^6}$$

    - CPI varies b/t programs on a given CPU
    - Don't account for
        - capabilities of instructions: cannot compare computers with different instruction sets
        - MIPS varies b/t programs on the same computer, thus a computer cannot have single MIPS rating
        - ISAs / IC
            - so if A: MIPS = 2, B: MIPS = 3, we cannot say B performs better than A
            - same application is compiled by x86 and ARM
            - #IC (x86) low, #IC (x86) higher, more instruction can be executed; however x86 performs better
            - only if their ISAs are the same

5. The Power Wall
    - Power = $Capacitive\ load * V^2 * frequency$

    - power: the cost for running electronic devices
    - Cannot reduce V further or remove more heat
    - Solution: switch to multicore microprocessors:

        - More than one processor per chip
    - single CPU included 2 cores:
        - P 正比于 # of cores(capacity load)
        - P = 2cv^f
    - 注意 如果 increase clock frequency to increase performance:
        - f → 2f, V 正比 f
        - P = c(2v)^2(2f) = 8cv^2f
        - 对比 2 cores in a single CPU

6. SPEC CPU Benchmark:
    - programs used to measure performance
    - develops benchmarks for CPU, I/O, Web
    - SPEC CPU 2006
        - CINT2006(integer)
        - CFP2006(floating-point)

   SPEC Power Bench mark:
    - Even Load% = 0(no instruction is executed by the processor), it still consumes power
    - At 10% load, 121W(47% of full load)

7. Moore's Law
    - **Number of transistors (in a dense integrated circuit) double every 2 years**
    - transistor: an on/off switch controlled by an electronic signal

Ch.2 Instructions: Language of the Computer

1. ISA: Instruction Set Architecture

1. ISA defines the CPU or CPU family
    - includes CPU view of memory, registers, number and roles
    - the contract between software and hardware
    - e.g. a program can be compiled by x86 compiler / ARM complier
    - compiled program = assembly has different instructions; cannot understand each other
    - ISA defines instruction, registers, addressing…of computer system
    - CPU architecture designs the micro computer hardware, and ≠ ISA

2. RISCV ISA

1. x86 ISA: designed for high-performance computers e.g. laptop / desktop
    - CISC-type
    - complex
2. ARM ISA: designed for smartphone/ tablet
    - RISC-type
    - simpler instruction decoder
    - simpler hardware
    - lower power consumption

- RISCV ISA
  - designed for research / education / commercial
  - Simple
  - modular

3. Operands of Computer Hardware

1. RISCV has a 32 * 64 bit register file
   - doubleword: 64-bit
   - 32 entries, width of each entry: 64 bit
   - one register size: 32-bit for RV32I; 64-bit for RV64I
2. Memory is byte addressed
   - differ by 8(bytes)
   - little endian
3. Register v.s. Memory
   - registers are faster to access than memory
   - make the common case fast→ immediate operands

4. Signed and Unsigned Numbers

1. unsigned: $0 \sim 2^N - 1$
2. signed: $-2^{N-1} \sim 2^{N-1} - 1$
3. signed:
   - -1: 111111….11111
   - most negative: 10000….000
   - most positive: 011111….111

     2: 000000….10
     -2: 反转+1
   - Sign extension: replicate the sign bit to the left
     - +2 00000010=> 0000000000000010
     - -2: 11111110=>1111111111111110
     - lbu: load byte unsigned
     - lb: load byte(signed)

5. RISC-V instructions

1. 32 bit
2. R-type: for register
3. I-type: used by arithmetic operands with one constant operand
   - addi
   - doesn't have subi because immediate field represents a two's complement integers
   - ld
   - **12 bit immediate:** -2^11~2^11-1
   - can refer to any doubleword(2^8) why? doubleword = 8 bytes of the base address in the base register rd
   - S-type:
     - immediate split into 2 parts: 7+5 **(still 12 bit)**

- in order to keep rs1, rs2, funct3, opcode fields in the same place in all instruction formats(similar formats reduce hardware complexity)

Instruction	Format	funct7	rs2	rs1	funct3	rd	opcode
add (add)	R	0000000	reg	reg	000	reg	0110011
sub (sub)	R	0100000	reg	reg	000	reg	0110011

Instruction	Format	immediate	rs1	funct3	rd	opcode
addi (add immediate)	I	constant	reg	000	reg	0010011
ld (load doubleword)	I	address	reg	011	reg	0000011

Instruction	Format	immed -iate	rs2	rs1	funct3	immed -iate	opcode
sd (store doubleword)	S	address	reg	reg	011	address	0100011

6. Stored-program concept:

1. instructions are represented as numbers
2. programs are stored in memory to be read or written, just like data
3. → if CPU want to execute instructions(stored in main memory) → need to get from main memory to instruction memory inside CPU

(MAIN) Memory can contain:

4. Accounting program (machine code)
5. Editor program (machine code)
6. C compiler (machine code)
7. Payroll data
8. Book text
9. Source code in C for editor program

processor need to know the address:

use PC: program counter == address of instructions

7. Logical operations

1. slli: shift left logical immediate
   - slli x11, x19, 4 // reg x11 = reg x19 << 4bits
   - sll
2. srli: shift right logical immediate
   - shift instructions use I-type format
   - only can shift <= 64 bit → **immediate = 6 bit (unsigned only)**
   - srl
3. and, andi
4. or, ori
5. xor, xori

- xor: =1 only when two values are different
- NOT→ xor 111…..111

6. one way to implement NOT is to use XOR with one operand being all ones(FFFFFFFFFFFFFFFF hex)

8. SB-type (conditional branch format)

1. beq: branch if equal
   - beq rs1, rs2, L1(100)
2. bne: branch if not equal
   - bne rs1, rs2, L1(100)
3. blt: branch if less than

4. bge: branch if greater or equal

5. bltu: branch if less, unsigned

6. bgeu: branch if greater/ equal unsigned

7. use beq x0, x0, Exit to go to the end of if statement (unconditional branch that the processor always follow the branch→ use a conditional branch whose condition is always true)

8. use beq x0, x0, Loop to branch back to the while test at top of the loop

9. **Basic blocks:**
   - **a sequence of instructions with**
     - **no embedded branches except at end**
     - **no branch targets except at beginning**
   - branch instructions included in machine code, degraded performance
   - sequential instructions all executed once start
   - 1 instruction can be executed every clock
   - larger basic blocks → reduce CPI thus performs better

9. Case/Switch Statement

1. indirect jump instruction: unconditional branch to the address specified in a register
2. procedure call: jump and link
   - jar x1, ProcedureAddress // jump to PA and write return address to x1
   - jal actually saves PC+4 in its designation register(x1) to link t

   - jalr x0, 0(x1)
   - branches to the address stored in register x1 and return 0

   - The calling program(the caller) puts the parameter values in x10-x17 and uses jar x1, X to branch to procedure X (the callee)

- The callee performs all the calculations and places the results in the same parameter registers, and return control to the caller sing jalr x0, 0(x1)

- ** jal x0, Label // can be used unconditionally branch to Label : discard the return address

10. Using more registers

1. stack pointer: x2 / sp 8 bytes
2. adjusted by one doubleword for each register
3. stacks grow from higher addresses to lower addresses
4. branch back to calling routine: jalr x0, 0(x1)
5. x5~x7 and x28~x31: temporary registers that are not preserved
6. x8-x9, x18-x27: saved registers that must be preserved on a procedure call
    - if used, the callee saves and restore them

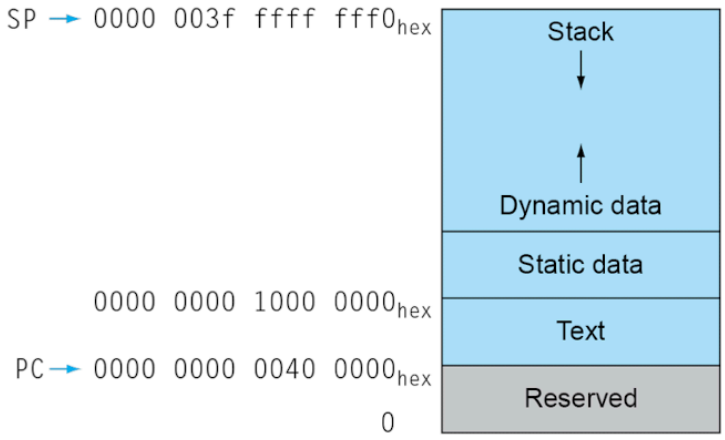
11. Nested procedure

1. leaf procedures: do not call others
2. nested procedure:
    - caller pushes any argument registers or temporary registers needed
    - callee pushes and return address register x1 and any saved registers used by callee

12. Memory Layout

- →SP

1. **Stack:** from higher address to lower addresses
2. **Dynamic data (heap):** from lower addresses to higher addresses e.g. malloc in C/ new in Java/ linked list (grow and shrink during lifetimes)
3. **Static data**: global variables; for constants and other static variables e.g. constant array(fixed length), strings
4. **Text segment:** home of the RISC-V machine code / program code
    → PC
5. reserved



13. Local data on the Stack

1. frame pointer: fp/ x8

2. FP: point to the first doubleword of the frame of a procedure
3. offer a stable base register within a procedure for local memory-references
4. during the procedure call:
    - FP →
        - Saved argument registers (if any)
        - Saved return address
        - Saved registers(if any)
        - Local arrays and structures (if any)

        SP→

        SP 在变化以后可以回到 FP 指向的位置

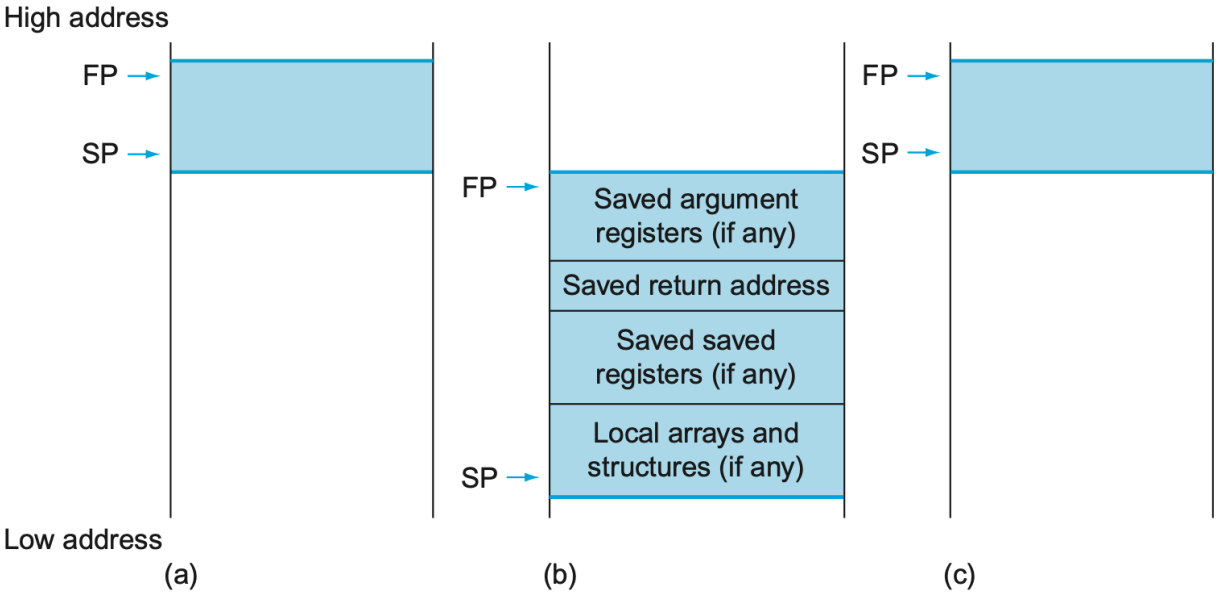


FIGURE 2.12 Illustration of the stack allocation (a) before, (b) during, and (c) after the procedure call. The frame pointer (`fp` or `x8`) points to the first doubleword of the frame, often a saved

14. Wide Immediate

1. lui: load upper immediate
2. lui rd, constant
    - copies [31: 12] 20 bits of rd
    - leftmost 32 bit: filled with copies of bit 31
    - rightmost 12 bits are filled with zeros
    - 只需要用 lui 搬运 imm 的[31:12]
    - 再用 addi 搬运前[11:0]就可以了

15. Branch Addressing

1. branch- SB-type: immediate [12:1] in total 12
2. branch address from [-2^11, 2^11-1] * 2 = [-2^12, 2^12-2] 中的双数地址
3. jump-and-link(jal) only instruction that uses UI-type format
    - 20-bit immediate [20:1]
    - address rage: [-2^19, 2^19-1]*2

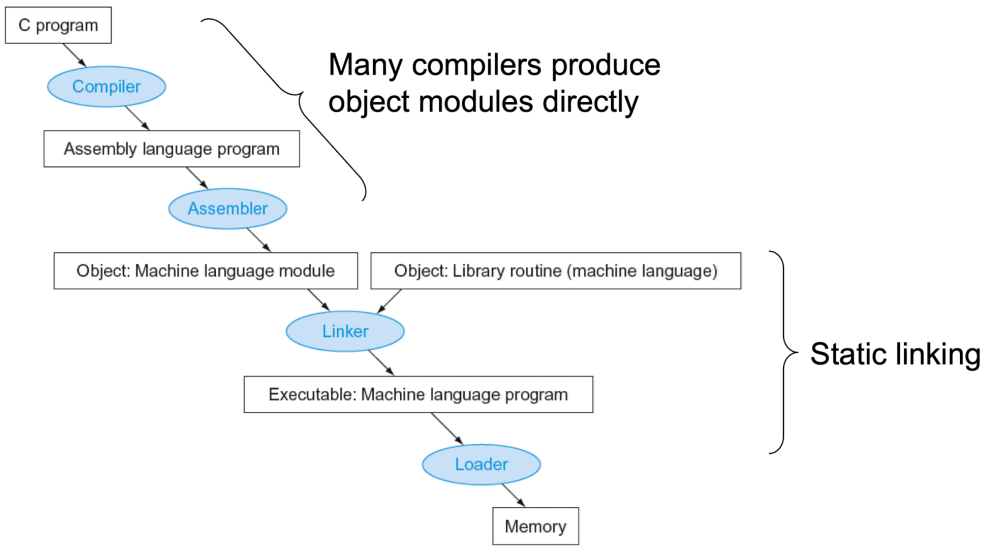
- / 4. = words: [-2^18, 2^18]
- RISC-V uses PC-relative addressing for both conditional branches and unconditional jumps
- but limited to 2^18 words
- Hence RISC-V allows very long jumps to any 32-bit address with 2 instruction-sequence:
    - lui x5, constant // constant = [31:12]
    - jalr x0, imm(x5) // imm = [11:0]
- jump-and-link-register(jalr) – UJ format
    - imm[20:1]

Name (Field Size)	Field 7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	Comments
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format

16. RISCV Addressing Mode

1. Immediate addressing : op = a constant
    - addi
2. Register addressing: op = a reg
    - add
3. Case addressing: op = a constant , address = reg + constant
4. PC-relative addressing: branch address = PC + constant

17. Translating and Starting a program



- A HLL program first compiled into an assembly language program and then assembled into an object module in machine language.
    - x.c
    - x.s

        - object file: x.o
        - statically linked library routines: x.a
        - dynamically linked library routes: x.so (oly link/ load library procedure when it is called)
    - Linker: combines multiple modules with library routines to resolve all references
        - executable file: a.out
    - Loader: places machine code into the proper memory locations for execution by the processor

Ch.3 Arithmetic for Computers

1. Addition and Subtraction
    - overflow occurs
    - 同号相加，结果异号
    - + + + = −
    - − + − = +
    - + − − = −
    - − − + = +
    - compiler can check for unsigned overflow using a branch instruction
    - ALU: arithmetic logic unit

2. Mul 64bit* 64bit <= 128bit
    - mul: multiply
        - lower 64 bits of the product
    - mulh: multiply high
        - upper 64bits, signed op
        - used to check 64bit overflow
    - mulhu: unsigned high
        - upper 64 bits, unsigned op
    - mulhsu: multiply high signed / unsigned
        - one op is signed and the other is unsigned
    - can be pipelined
        - several multiplication performed in parallel
3. Division
    - cannot use parallel
        - subtraction is conditional on sign of remainder
4. Floating point numbers

    - Two representations
        - single precision: float 32 bit
        - double precision: double 64 bit

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

$$(-1)^{S} \times (1 + (s1 \times 2^{-1}) + (s2 \times 2^{-2}) + (s3 \times 2^{-3}) + (s4 \times 2^{-4}) + \ldots) \times 2^{E}$$

        - single: 8-bit exponent+ 23-bit fraction; bias = 127 = 2^7 -1
        - double: 11 + 52 ; bias = 1023 = 2^10-1
        - actual exponent = exponent – bias
    - single-precision range
        - exponent: 8-bit; but 00000000 and 11111111 reserved
        - smallest: 00000001
            - actual exponent = exponent – bias = 1-127 = -126
            - Fraction: 0000…000 significand = 1.0
            - +/- 1.0* 2^(-126)
        - largest: 11111110
            - actual exponent = exponent – bias = 254-127 = 127
            - Fraction: 1111….111 significand ~~ 2.0
            - +/- 2.0* 2^(127)
    - double-precision range
        - exponent: 11 bit
        - smallest: 000…….001
            - 1-1023 = -1022
            - +/- 1.0*2^(-1022)
        - largest: 1111….0
            - 2046-1023 = 1023
            - +/- 2.0*2^1023
    - relative precision
        - single: approx. 2^(-23)
            - 23*0.3~6 decimal digits of precision
        - double: approx.. 2^(-52)
            - 52*0.3~ 16 decimal digits of precision
    - Denormal numbers
        - exponent = 0…..0000, fraction = 0
            - x = (-1)^S *(0+0)* 2^(0-Bias) = +/- 0.0
        - exponent = 111….1111 (255 / 2047) , fraction = 0…0 (0)
            - +/- infinity
        - exponent = 111….1111(255 / 2047), fraction = nonzero
            - NaN (Not a Number)
    - Sperate FP register: f0….f31 (64 bit)
        - double-precision
        - single-precision values stored in the lower 32 bits
        - FP instructions operate only on FP registers
            - faad.s f0, f1, f2 // f0 = f1+f2
            - fsub.s
            - fmul.s
            - fdiv.s
            - fadd.d …..
            - flw f0, 4(x5) // f0 = Memory[x5+4]
            - fld f0, 8(x5) // f0 = Memory[x5+8]
            - fsw f0, 4(x5) // Memory[x5+4] = f0
            - fsd f0, 8(x5) // Memory[x5+8] = f0

- fallacies and pitfalls
  - shift right: only worked for unsigned
  - floating-point addition is not associative

Ch.4 The Processor(The Single-Cycle Processor)

1. CPU performance factors
   - Instruction Count:
     - determined by ISA and compiler
   - CPI and Cycle time
     - determined by CPU hardware (can generate 1ns [best] / 10ns/ 100ns…)

2. Instruction Execution
   - PC → assume instructions are stored in the instruction memory in this chapter
   - Register File: a collection of registers in which any registers can be read / written by specifying # of reg in this file
   - ALU: arithmetic logical unit used to calculate
     - arithmetic result : a/s/m/d/&/ | /^
     - memory address for load / store
     - branch comparison
   - Access data memory for load / store
   - PC: target PC / PC+4

3. Clocking methodology
   - combinational logic transforms data during clock cycles b/t clock edges
   - longest delay determines clock period

4. Datapath
   - elements(hardware components) that process data and addresses in the CPU

5. Reg
   - 2 regs for read: rs1, rs2
   - 1 reg for write: rd to specify the register number to be written
   - 1 write data
   - 2 read data outputs
   - reg: 5 bit
   - data input / data outputs: 64bit
6. Data Memory
   - ld / sd
   - calculate address using 12-bit offset
     - use ALU, but sign-extend offset
     - Immediate generation: 12 bit 2's complement → 64 bit
   - Load: Read memory and update register
   - Store: write register value to memory
   - 2 input: address, write data
   - output: read data

7. ALU
    - 2 64bit inputs
    - 1 64 bit outputs
    - 4 bit control signal
        - 0000 AND
        - 0001 OR
        - 0010 add
        - 0110 sub
8. Branch Instructions
    - PC
    - Read register operands from RF
    - Compare operands
        - Use ALU, subtract and check 0 output (Zero)
    - PC update
        - PC = PC+4
        - PC = PC(64 bit) + imm << 2 + sign-extend
9. MUX
    - 2 input, 1 output
    - controlled by control unit

10. Six control signals:

    - ALUSrc:
        - need imme value : 1
    - ALUOp
        - ld 00
        - sd 00
        - beq 01
            - ALUOp = 00 / 01 not depend on funct7 / funct3
        - R-type 10
    - Branch
        - 1: beq
    - MemRead
    - MemtoReg
        - 和 MemRead 同步 1 when LD
    - MemWrite
        - when SD
    - RegWrite:
        - only when R-type / ld

11. Path
    - Instruction memory
    - Reg file
    - ALU
    - Data memory
    - Reg file