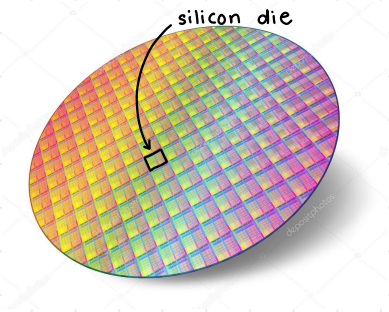
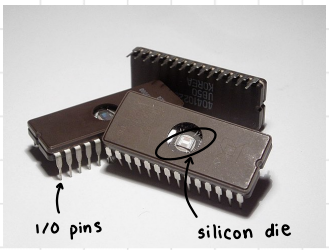
CHAPTER 1

INTRODUCTION

- **Moore's Law (1965)**: integrated circuit (IC) resources (e.g. # of transistors) doubles every 1.5-2 yrs
  - ↳ post-Moore era, newer 2018 chips utilize **7nm process technology**
    - 7nm is smallest feature size on chip
- below is **silicon wafer**



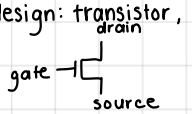
  - ↳ dies are put into individual pkg chips



can be put into printed circuit board

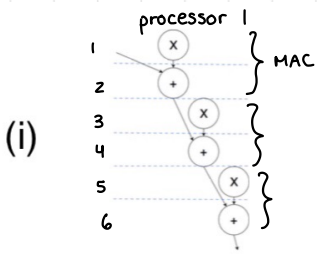
- silicon → wafer processing → test → die → test → pkg → customer

- **abstraction** is type of design idea
  - ↳ levels of design: transistor, gate, architecture, behaviour (data flow graph)
- **transistor**:

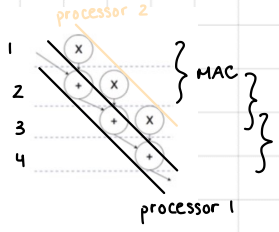


  - ↳ '0' — D, S
  - ↳ '1' — D, S

  - ↳ don't design w/ transistors
- **parallelism** is when an application does more than 1 process at the same time
  - ↳ e.g. consider digital filter computation shown below



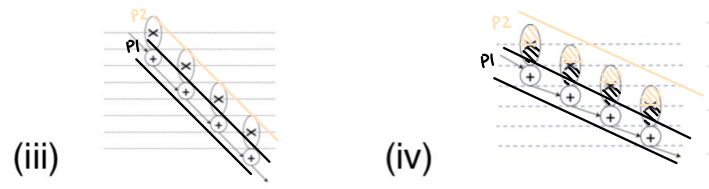
(i)



(ii)

    - **MAC (multiply-accumulate computation)**: multiplication operation gets added to prev accumulated sum
    - i) has 1 **ALU (arithmetic logic unit)**/processor ∴ can do 0.5 MAC/cycle
    - ii) has 2 ALUs ∴ can do 1 MAC/cycle
  - ↳ parallelism provides increased performance

- realistically, multiplication takes longer than addition so parallelism looks like iii)



(both are examples of parallelism)

↳ assume multiplication takes twice as long as addition
- iii) does 0.5 MAC/cycle
- iv) uses a pipelined multiplier & does 1 MAC/cycle
  - 1st half of multiplication is done by specific piece of hardware
  - 2nd half of multiplication is done by another piece of hardware
  - pipelining allows us to overlap multiplications

- other techniques to improve system performance:
  ↳ common case fast: e.g. if 90% of time happens in 10% code, optimize this code
  - typically done in assembly
  ↳ prediction: in conditional branching, start doing work on most likely outcome of branch to save time
  ↳ memory hierarchy: smaller memory is faster so it's placed closer to processor
  ↳ testable / reliable

TERMINOLOGY AND EQUATIONS

- hierarchy below program:
  1) application software: written in high-level language
  2) system software
     - compiler: translates HLL to machine code
     - operating system: service code
       - handles I/O
       - manages memory & storage
       - schedules tasks & shares resources
  3) hardware: processor, memory, I/O controllers
- levels of program code:
  ↳ high-level language
  - level of abstraction closer to problem domain
  - provides for productivity & portability
  ↳ assembly language
  - textual rep of instructions
  ↳ hardware rep
  - binary digits (i.e. bits) sitting on hard disk drive
  - encoded instructions & data
- to run a program:
  1) OS loads program
  2) OS starts program execution
     - process begins
- same components for all kinds of computers (e.g. desktop, server, embedded)
  ↳ I/O includes:
  - user-interface devices (e.g. display, keyboard, mouse)
  - storage devices: hard disk, CD/DVD, flash

- these are non-volatile (when power's off, data stays)
  - network adapters: for communication w/ other computers
- ↳ processing (CPU)
- ↳ storing (memory)

- central processing unit (CPU) is split into datapath (e.g. ALU, busses, registers) & control (e.g. datapath commands, memory, I/O)
- there are many types of memory
  - ↳ random access memory (RAM) is volatile
    - static RAM
    - dynamic RAM (dominates main memory)
    - resistive RAM
    - magnetic RAM
  - ↳ cache: small & fast memory
    - buffer memory for DRAM chips
  - ↳ secondary memory (e.g. magnetic disks)
  - ↳ flash is non-volatile
    - wears out after 100k - 1M writes
    - slower & cheaper than DRAMs but smaller, more power-efficient, & costly than disks
  - ↳ read-only memory (ROM) is non-volatile
    - EEPROM is electrically erasable programmable ROM

- cost of IC:
  - ↳ $\text{cost per die} = \frac{\text{cost per wafer}}{\text{dies per wafer} \times \text{yield}}$
    - dies per wafer $\simeq$ wafer area / die area
    - $\text{yield} = \frac{1}{(1 + (\text{defects per area} \times \frac{\text{die area}}{2}))^2}$
- performance measures execution time from start to end (i.e. latency) & throughput / bandwidth (total jobs completed per period of time)
  - ↳ $\text{performance} = \frac{1}{\text{execution time}}$
  - ↳ X is n times faster than Y
    - $\frac{\text{performance X}}{\text{performance Y}} = \frac{\text{execution time Y}}{\text{execution time X}} = n$
    - i.e. Y is n times slower than X
    - i.e. X is n times as fast as Y

- Amdahl's Law states execution time after improvement = $\frac{\text{execution time affected by improvement}}{\text{amount of improvement}}$ + execution time unaffected
- CPU execution time doesn't include time waiting for I/O or in OS waiting for task to be performed by CPU
  - ↳ clock frequency: kHz or MHz
  - ↳ CPI: avg # of cycles per instruction (avged over all instructions in program)
  - ↳ $\text{MIPS} = \frac{\text{clock rate}}{\text{CPI} \cdot 10^6}$
    - millions of instructions per second
  - ↳ CPU execution time $= (\#\text{ of clock cycles})(\text{clock period})$
    $= \frac{\#\text{ of clock cycles}}{\text{clock freq}}$
    $= \frac{(\text{instruction count})(\text{CPI})}{\text{clock freq}}$
- dynamic power of circuit is proportional to $\frac{1}{2}CV^2f$
  - ↳ C is capacitance
  - ↳ V is power supply (Vdd)

↳ f is clk freq

- **static power** of circuit comes from leakage, i_L
  ↳ i_L inc as Vdd dec
- to dec dynamic power, easiest way is to reduce Vdd but inc leakage current
  ↳ results in power dissipation

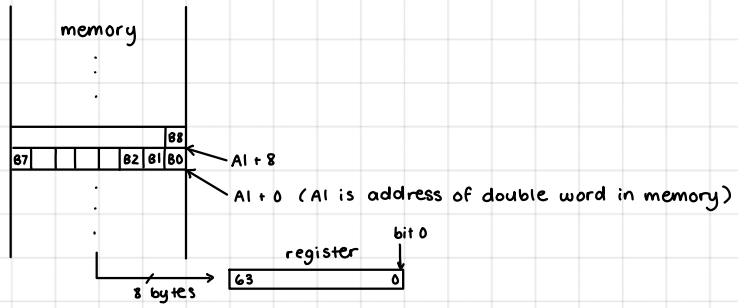
- **RISC : reduced instruction set computer**
  ↳ diff from CISC (complex instruction set computer)
- in course, we'll use RISC-V core
  ↳ **coprocessor** : unit attached to RISC-V core & mostly sequenced by RISC-V instruction stream (i.e. programmable) but has additional state & instruction set extensions
  ↳ **accelerator** : non-programmable fixed-function unit/core that operates autonomously but is specialized for certain functions
    - e.g. I/O accelerators offload I/O tasks from main app cores
  ↳ software running on RISC-V can run at **privilege levels**:
    - M-mode : machine mode (highest privilege)
    - U-mode : user mode
      - conventional app
    - S-mode : supervisor mode
      - OS usage

CHAPTER 2

INSTRUCTION SET ARCHITECTURE

- ISA defines interface btwn hardware & software
- RISC-V instructions are 32-bit
  - ↳ RISC-V ISA is family of instructions:
    - RV32I (32 reg)
    - RV32E (16 reg)

      RISC-V — width of registers (address space size) — embedded
    - RV64I (32 reg)
    - RV128I (32 reg)
  - ↳ we'll use RV32I & RV64I
- RISC-V assembly language notation uses 64-bit registers
  - ↳ 64 bits is double word
  - ↳ 32 bits is word
  - ↳ 8 bits is byte
- there's 32 registers (x0 - x31)
  - ↳ x0 value is always 0
- to perform arithmetic operations, data must always be in registers
  - ↳ since # vars in programs are usually > 32, less/later used vars are spilled into memory (spilling registers)
    - registers are faster & more energy efficient than memory
- memory is byte addressable
  - ↳ RV64I (double word)



  - ↳ convention is little endian, meaning that address of 64-bit double word refers to address of rightmost byte (LSB) which contains bit 0
    - sequential double word accesses differ by 8
  - ↳ RV64I has 2^{61} addressable double words
    - 3 bits are used for byte addressing since there's 8 bytes in a double word
  - ↳ RV32I has 2^{30} addressable words
    - 2 bits are used for byte addressing since there's 4 bytes in a word
- program counter (PC) register holds address of current instruction being executed by processor

  some notation for machine code:
  - ↳ R[rs1] is contents of reg rs1
  - ↳ M[addr] is value stored at address addr in memory
  - ↳ {imm, 16'0} denotes concatenating immediate (imm) w/ 1 bit of value 0
    - {imm, 12'b'0} uses 12 zero bits
- data memory is byte addressable
  - ↳ data is 64 bits so address + 8 to get next data value

- instruction memory is byte addressable
  - ↳ instructions are 32 bits so PC + 4 to get next instruction

RISC-V INSTRUCTIONS

- instructions have similar instruction format
  - ↳ opcode fields: opcode, funct3, funct7
  - ↳ source registers: rs1, rs2
  - ↳ destination register: rd
  - ↳ immediate field: imm
- to translate assembly language instructions into machine language, look at RISC-V ref card
- **R-type instructions** use only registers
  - ↳ coded in machine language as:

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5bits	3 bits	5 bits	7 bits

  - ↳ usually used for arithmetic instructions using reg
    - e.g. add x5, x6, x2 → 0000000 00010 00110 000 00101 0110011
      - → verilog description for add: `R[rd] = R[rs1] + R[rs2]`
  - ↳ e.g. d = b + c - e in assembly language (b: x6, c: x2, d: x3, e: x4)

```
add x5, x6, x2   // a is temp var in x5
sub x3, x5, x4
```

- **I-type instructions** use immediates
  - ↳ coded in machine language as:

immediate[11:0]	rs1	funct3	rd	opcode
12 bits	5bits	3 bits	5 bits	7 bits

    - immediate 12-bit value has to be sign extended from 12 bits to 64-bit double word size in RISC-V
      - → e.g. sign extension of 1111 1111 1000 to 64-bits is 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1000
  - ↳ used for loads & immediate arithmetic
    - e.g. ld x9, 64(x22) → 0000 0100 0000 10110 011 01001 0000011
      - → verilog description: `R[rd] = M[R[rs1] + imm](63:0)`
  - ↳ e.g. g = h + A[8] in assembly language where A[] is array of 100 words w/ starting/base address in x22 (g: x20, h: x21)

```
ld x9, 64(x22)   // load double word A[8] into x9
                 // RISC-V is byte addressed so offset is 64 = 8 · 8 since there's 8 bytes in
                 // a double word
add x20, x21, x9
```

- **S-type instructions** means they store values in memory
  - ↳ coded in machine language as:

immediate[11:5]	rs2	rs1	funct3	immediate[4:0]	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

    - 12-bit immediate split is split to keep location of rs1 & rs2 fields consistent
  - ↳ e.g. storing double word in x9 into A[30] (base address of A[] is in x10) is sd x9, 240(x10) → 0000111 01001 01010 011 10000 0100011
    - verilog description: `M[R[rs1] + imm](63:0) = R[rs2](63:0)`
  - ↳ e.g. A[30] = h + A[30] + 1 in assembly language where base address of A[] is

stored in x10 (h: x21)

```
ld  x9, 240(x10)    // x9 = A[30]
add x9, x21, x9     // x9 = h + x9
addi x9, x9, 1      // x9 = x9 + 1
sd  x9, 240(x10)    // A[30] = x9
```

- **SB-type instructions** are for conditional branching
  - ↳ coded in machine language as:

immediate[12,10:5]	rs2	rs1	funct3	immediate[4:1,11]	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

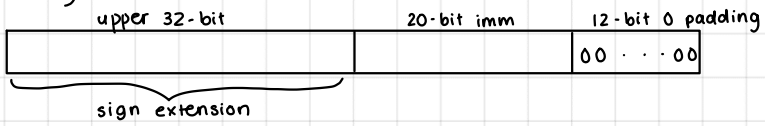
  - ↳ e.g. if x0 != x7, keep looping

```
    addi x7, x0, 5
loop:
    addi x7, x7, -1
    bne  x0, x7, loop   } PC - 4
                        } PC + 4
```

    - verilog description: `if (R[rs1] != R[rs2]) PC = PC + {imm, 1'b0}`
    - to add -4 as imm: 0100 (+4) → 2's complement → 1011 + 1 = 1100 → 1111 1111 1111 1100
    - in machine language, 1111111 00111 00000 001 11101 1100011
- {, } means concatenation
- Fr'n
  - ↳ F: # binary bits
  - ↳ r: radix (e.g. b, h, d)
  - ↳ n: digits
  - ↳ e.g. 32'b0 is 32 binary zero digits
- **U-type instructions** are useful for loading large imm values into reg
  - ↳ coded in ML as:

immediate[31:12]	rd	opcode
20 bits	5 bits	7 bits

  - ↳ e.g. lui stands for load upper immediate
    - verilog des is `R[rd] = {32'b imm<31>, imm, 12'b0}`



    - upper 32 bits takes MSB of imm & does sign extension
      - → i.e. 32'b imm<31>
- **UJ-type instructions** are for unconditional jumps
  - ↳ coded in ML as:

immediate[20,10:1,11,19:12]	rd	opcode
20 bits	5 bits	7 bits

  - ↳ e.g. jal (jump & link) is usually used for procedure calls
    - link means to return to where it was called by placing return address (link) into specified register
    - `jal x11, 2000  // go to location 2000`

- verilog descrip: `R[rd] = PC + 4 ; PC = PC + {imm, 1'b0}`
  - link: return address
  - jump
  - ↳ longer branches w/more reach can be achieved since imm is 20 bits instead of 12
- similar pos for imm data to minimize hardware complexity:

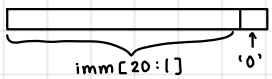
	funct7		rs2		rs1, funct3	rd		opcode
I	imm[11]	**imm[10:5]**	**imm[4:1]**	imm[0]				opcode
SB	imm[12]	**imm[10:5]**				**imm[4:1]**	imm[11]	opcode
S	imm[11]	**imm[10:5]**				**imm[4:1]**	imm[0]	opcode
U	imm[31]	imm[30:25]	imm[24:21]	imm[20]	**imm[19:12]**			opcode
UJ	imm[20]	**imm[10:5]**	**imm[4:1]**	imm[11]	**imm[19:12]**			opcode

PROCEDURES

- by convention in RISC-V:
  - ↳ 8 parameter registers x10 - x17 are for passing parameters or returning values
  - ↳ 1 return address register x1 which holds address to return to point of origin
- to call procedures:
  - ↳ jal branches to address of procedure & saves address of instruction after procedure call in x1
    - UJ-type
  - ↳ jalr jumps to address stored in register & is used at end of procedure to get back to where it was called from
    - I-type
  - ↳ e.g.

```
main:
    :
    jal x1, proced      → PC + 4
    :
proced:
    :
    jalr x0, 0(x1)
```

    - address after jalr is saved to x0 (it won't be saved) b/c we'll nvr need to go back to after it's been called
- branch has reach of $\pm 2^{10}$ words
  - ↳ 

  - ↳ 13 - 2 - 1 = 10 bits
    - byte address
    - sign
- jump has reach of $\pm 2^{18}$ words
  - ↳ 

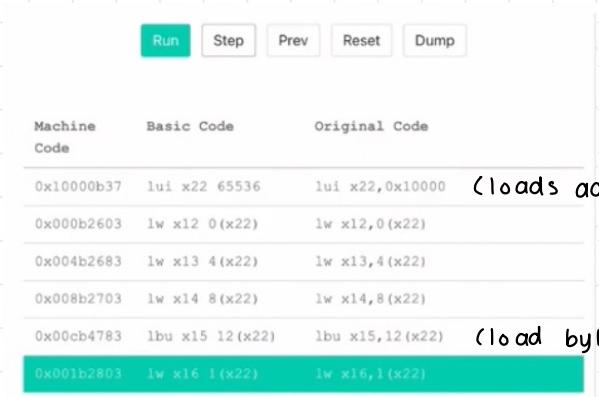
  - ↳ 21 - 2 - 1 = 18 bits
- e.g. to get longer reach, replace `beq x10, x0, L1` w/

```
bne x10, x0, L2
jal x0, L1
L2:
```

- in Venus memory, data is stored starting at address 0x1000 0000
  - ↳ .byte & .word are instructions that store data into memory

↳ .text signifies beginning of actual assembly code

- RISC-V supports misaligned address access
  ↳ e.g.



(loads address of memory into x22)

(load byte unsigned)

  - memory:

Address	+0	+1	+2	+3
x1000 0004	11	22	33	44
x1000 0000	0a	0b	0c	0d

  - lw x16, 1(x22) will load 0x110d0c0b in x16

- Venus only supports signed immediates
  ↳ e.g. 0xeef would be -0x111
- e.g. design RV321 code for partial code: while (A[x5] == x6) x5 = x5+1,
  ↳ assume base address of array A[] is in x19
  ↳ solution:

```
Loop: slli  x9, x5, 2     //shifting left by 2 is same as mult by 4 (word is 4 bytes in RV321)
      add   x9, x9, x19
      lw    x9, 0(x9)
      bne   x9, x6, Exit
      addi  x5, x5, 1
      beq   x0, x0, Loop
Exit:
```

RISC-V CONVENTIONS

- convention is required when working in teams
- return address register is x1
  ↳ can also directly use ra
- incoming parameters are in x10 - x17
  ↳ i.e. a0 - a7
- saved registers are in x19 - x22
  ↳ i.e s3 - s6
  ↳ won't be destroyed by any routine/procedure call
- temp registers are in x5 - x7
  ↳ i.e. t0 - t2
  ↳ data will be destroyed by procedure calls
- for any temp registers needed in procedure, registers are spilled to memory & restored when procedure is finished
- stacks are used to spill registers
  ↳ stack pointer is x2
    - i.e. sp
  ↳ stack grows from higher to lower addresses

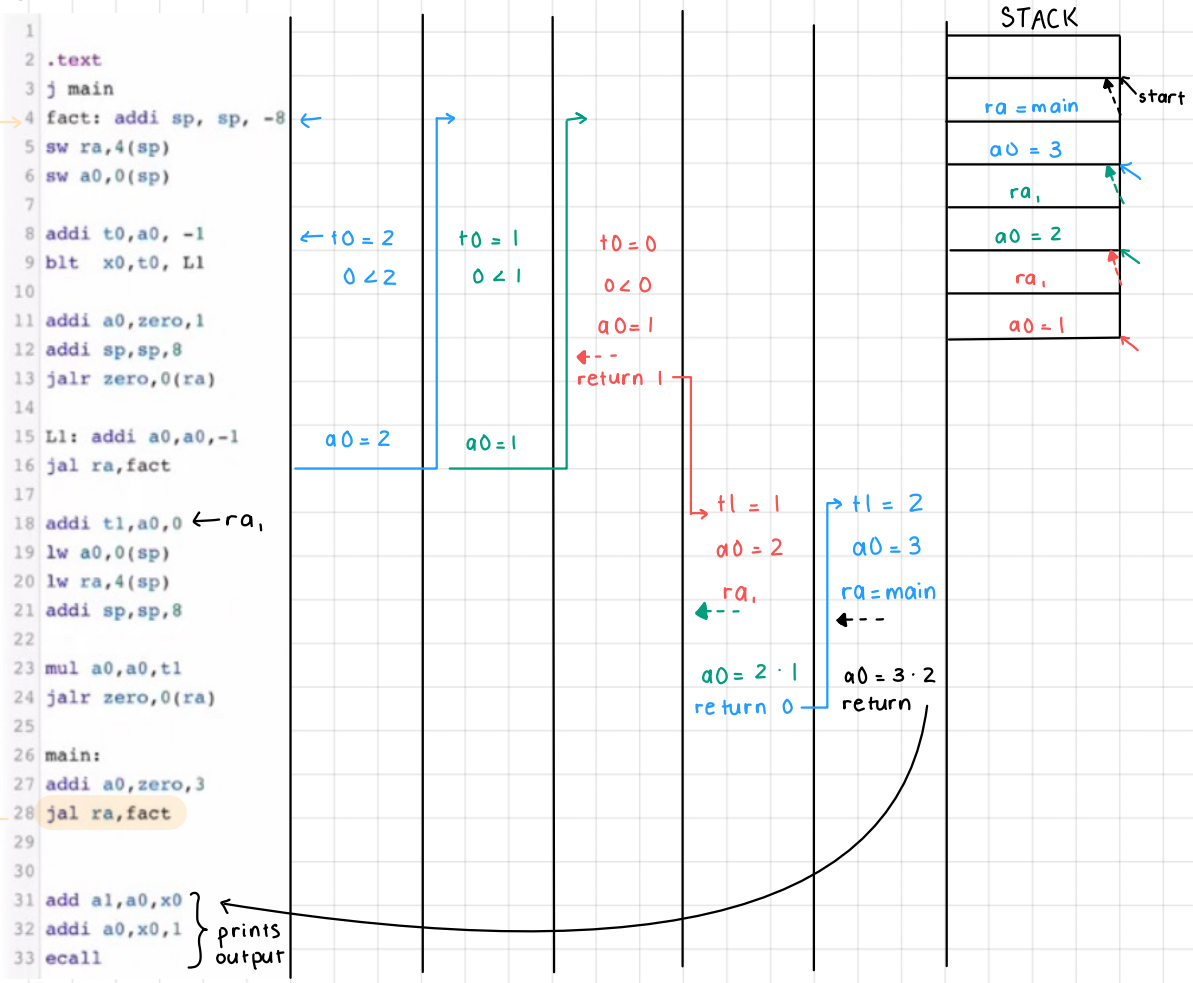
RISC-V separates 19 registers into 2 groups:

1) x5 - x7 & x28 - x31 are not preserved by callee so if caller needs these values, must push onto stack before proced call

2) x8 - x9 & x18 - x27 must be saved & restored by callee if they are used
   ↳ callee must also save ra / x1

- values are pushed on by decrementing sp & popped off by incrementing sp
  - ↳ e.g. proc:
```
addi sp, sp, -24      // proc needs to use x5, x6, x20
sd x5, 16(sp)
sd x6, 8(sp)
sd x20, 0(sp)
  :
ld x20, 0(sp)         //pop / restore 3 values
ld x6, 8(sp)
ld x5, 16(sp)
addi sp, sp, 24
jalr x0, 0(x1)        //return to where proc was called
```

- e.g. Venus code (RV32I) for factorial



- types of addressing:
  - ↳ immediate: operand is constant within instruction
    - e.g. imm12 for addi, imm20 for lui
  - ↳ register: operand is register
    - e.g. rs1, rs2, rd fields for add & sub
  - ↳ base addressing: operand is at mem location whose address is sum of register & constant
    - e.g. sw, sd
    - M[R[rs1] + imm] where rs1 contains base address
  - ↳ PC-relative: branch address is sum of PC & constant
    - PC = PC + {imm, 1'b0}
    - e.g. blt, beq
    - important for position independent code (PIC)
- RISC-V mem allocation for program & data:

addresses	memory contents
SP => 0000 003f ffff fff0	Stack || \/ /\ || Dynamic Data (Heap)
GP ⇒ 0000 0000 1000 0000	Static Data (constants, global variables)
PC=> 0000 0000 0040 0000	machine code "text segment"
0	reserved (e.g. I/O)

↳ global pointer (GP) points at static data, which typically sits above machine code

TOOLS

- process of compiler: input C program → file.c → file.s (assembly)
  - ↳ assembler makes an object file: file.s → file.o
  - ↳ linker resolves all undefined labels & outputs executable file: file.o → a.out
    - since there may be more than one file, linker has to solve data collision issues from 2 files storing diff values in same mem locations, deal w/ proced calls from file A to routines in file B, etc.
  - ↳ loader allows user to run exe file
- object file for Unix consists of 6 parts:
  - ↳ object file header: size & pos of other pieces in file
  - ↳ text segment: machine language code
  - ↳ static data segment: data allocated for life of program
    - static stays constant but dynamic can be resized
  - ↳ relocation info: identifies instructions/data dependent on absolute addresses
  - ↳ symbol table: labels are associated w/ addresses
    - remaining are undefined labels (i.e. external ref)
  - ↳ debugging info: how modules were compiled for debugger
- executable file is same as object but w/ generally no unresolved ref
  - ↳ exception is partial linked files (e.g. library routines) w/ unresolved address, which results in another object file
  - ↳ linker resolves all undefined labels using relocation info & symbol table for each obj module
- e.g.

object file header	name text size data size	**procedure A** 0x100 0x20	
text segment	address	instruction	
	0 4 …	ld x10, 0(x3) jal x1, 0 ↖ placeholders …	ld X call procedure B
data segment			
	0 …	X …	
relocation information	address	instruction type	dependency
	0 4	ld jal	X B
symbol table	label	address	
	X B		

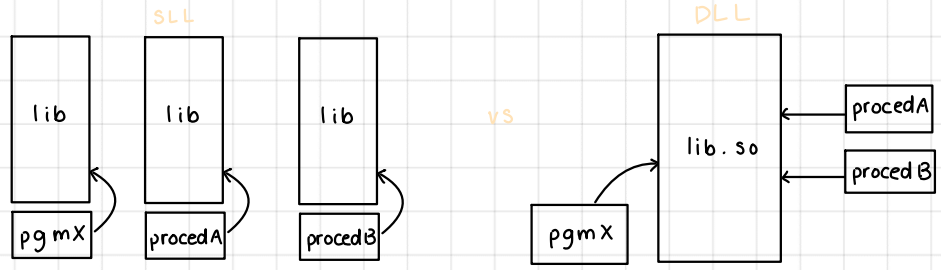
object file header	name text size data size	**procedure B** 0x200 0x30	
text segment	address	instruction	
	0 …	sd x11, 0(x3) …	sd Y
data segment			
	0 …	Y …	
relocation information	address	instruction type	dependency
	0 …	sd …	Y …
symbol table	label Y	-	

↳ executable file:

	text size data size	0x300 0x50
text segment	address	instruction
	0000 0000 0040 0000 0000 0000 0040 0004 … 0000 0000 0040 0100 …	ld x10, 0(x3) jal x1, 252_{10} … sd x11, 32_{10}(x3) ….
data segment	address	
	0000 0000 1000 0000 … 0000 0000 1000 0020 …	X value … Y value …

DYNAMICALLY LINKED LIBRARIES

- libraries usually are not **statically linked** b/c:
  - ↳ routines are part of exe so newer versions aren't incorporated automatically
  - ↳ all routines are loaded even if they're nvr used
  - ↳ each file has its own version of library (can be very large)
- **dynamic linking** means they're loaded when program is run
  - ↳ uses shared object file (e.g. library.so)
- SLL vs DLL:



lazy linkage is form of dynamic linking where library routine is linked only after it's called

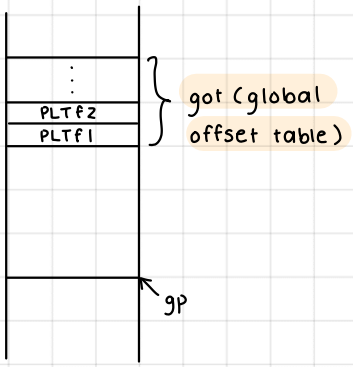
↳ e.g. procedA:

```
        ⋮
jal ra, libfl
        ⋮
jal ra, libfl
        ⋮
```

imm offset

```
libf1: ld  t3, got.f1(gp)
       jalr ra, t3
```

stub for libf1



PLTf1: 1) calls dynamic linker to find routine address & place it in got
2) calls libf1 so that we can correctly branch to routine
3) returns to calling site

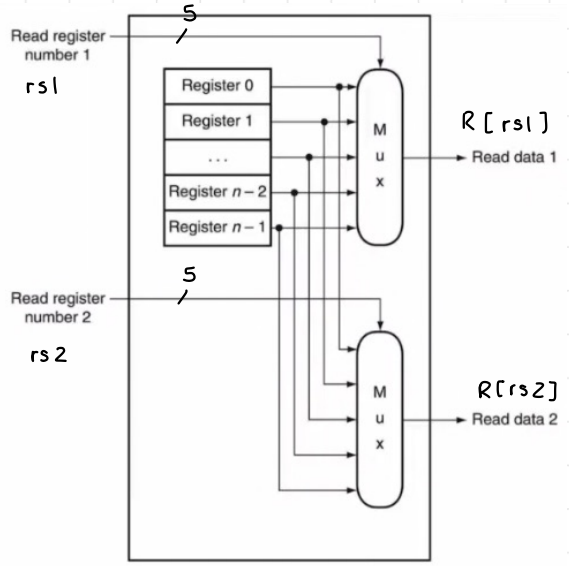
- PLT stands for procedural linkage table
  - ↳ PLT only needs to be called once for 1st time routine is called in order to place correct value in got
    - after, code jumps indirectly to routine w/o overhead since got.f1 value is alr correct

- for larger offsets, can use auipc (32-bit imm) instead of jal (20-bit imm)
  - ↳ e.g.
```
auipc t3, imm[31:12]
ld    t3, imm[11:0](t3)
jalr  ra, t3
```

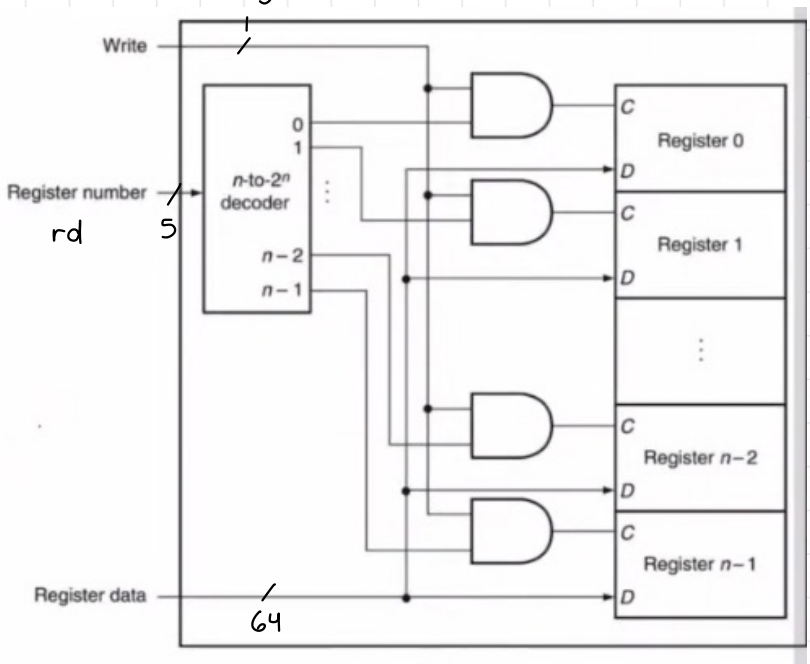
CHAPTER 4

HARDWARE IN PROCESSOR

- register file is array of processor registers in CPU
  - ↳ set of 32 reg for RISC-V
- to read data from reg file:



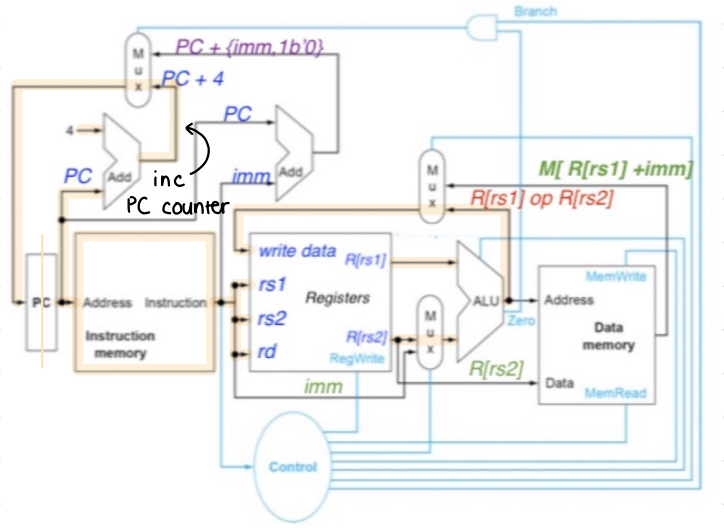
  - ↳ in RISC-V, use pair of 32-to-1 muxes ∴ 64 bits wide for RV64I
    - ° one mux for each bit
- to write data to reg file:



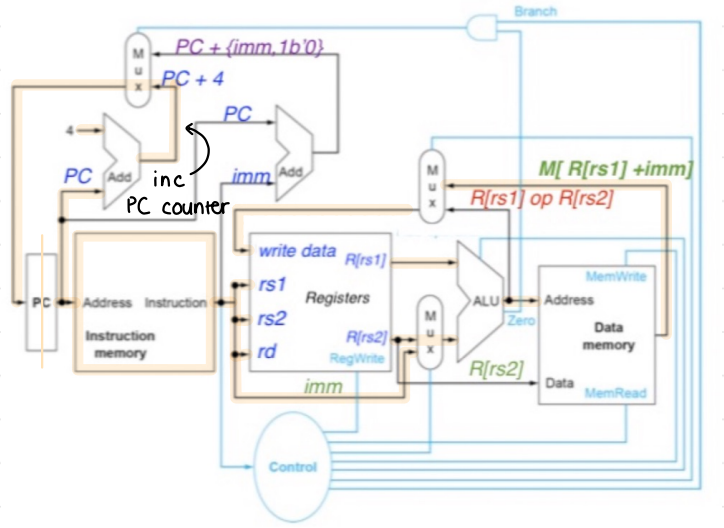
- 3 general types of ins to be supported by processor core:
  - ↳ mem load/store: ld, sd, etc.
  - ↳ arithmetic-logical ins: add, sub, and, or, etc.
  - ↳ conditional branch ins: beq, bne, etc.
- all 3 ins have following common functions:
  - ↳ use PC to fetch ins from mem
  - ↳ increment PC to point to next ins
  - ↳ read 1/2 reg values (rs1, rs2)

↳ perform ALU operation

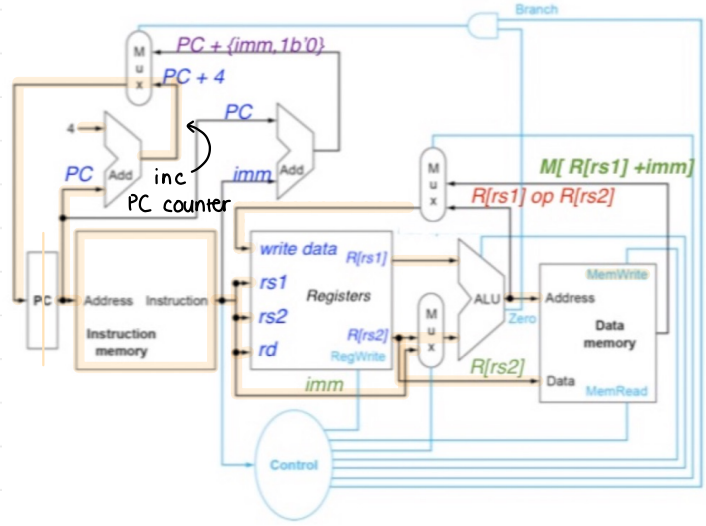
- hardware implementation will assume:
  - ↳ +ve edge-triggered clocking
  - ↳ PC & reg are implemented as +ve edge-triggered FFs w/ load/CE & reset signals
  - ↳ mem stores data on +ve edge
- add: R[rd] = R[rs1] + R[rs2] ●



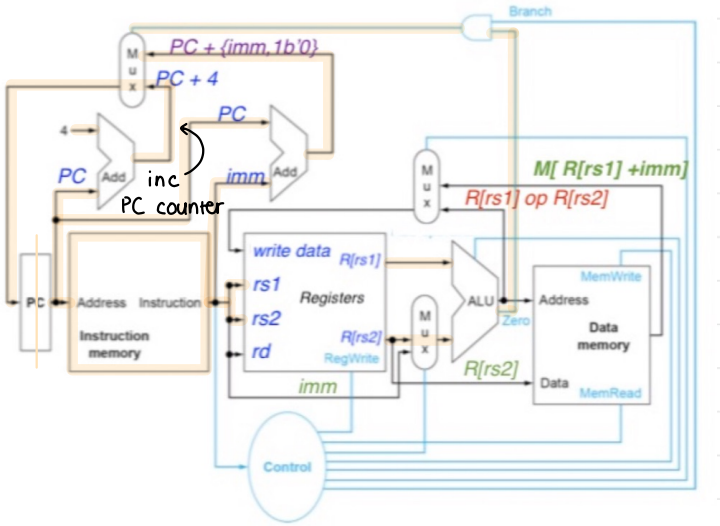
- ld: R[rd] = M[R[rs1] + imm] ●



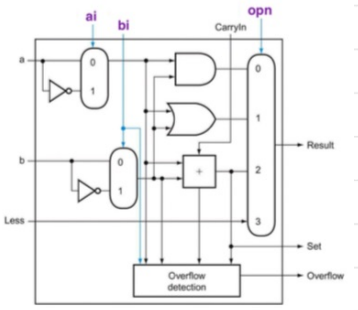
- sd: M[R[rs1] + imm] = R[rs2] ●



- beq : if R[rs1] == R[rs2] then PC = PC + {imm, 1'b0}



- control block processes opcode to decide which parts are needed for specified ins
  - ↳ 7 control signals:
    - Reg Write
    - ALUSrc (controls imm values being read into ALU)
    - ALUOp (2 bits)
    - MemWrite
    - MemRead
    - MemToReg
    - Branch
- assume all ins execute in 1 clock cycle
  - ↳ this is why ins & data mem units are separate
- control block is split into 2: control & ALU control
- ALU control block outputs 4 bits: ai (Ainvert), bi (Binvert), opn (operation)
  - ↳



  - ↳

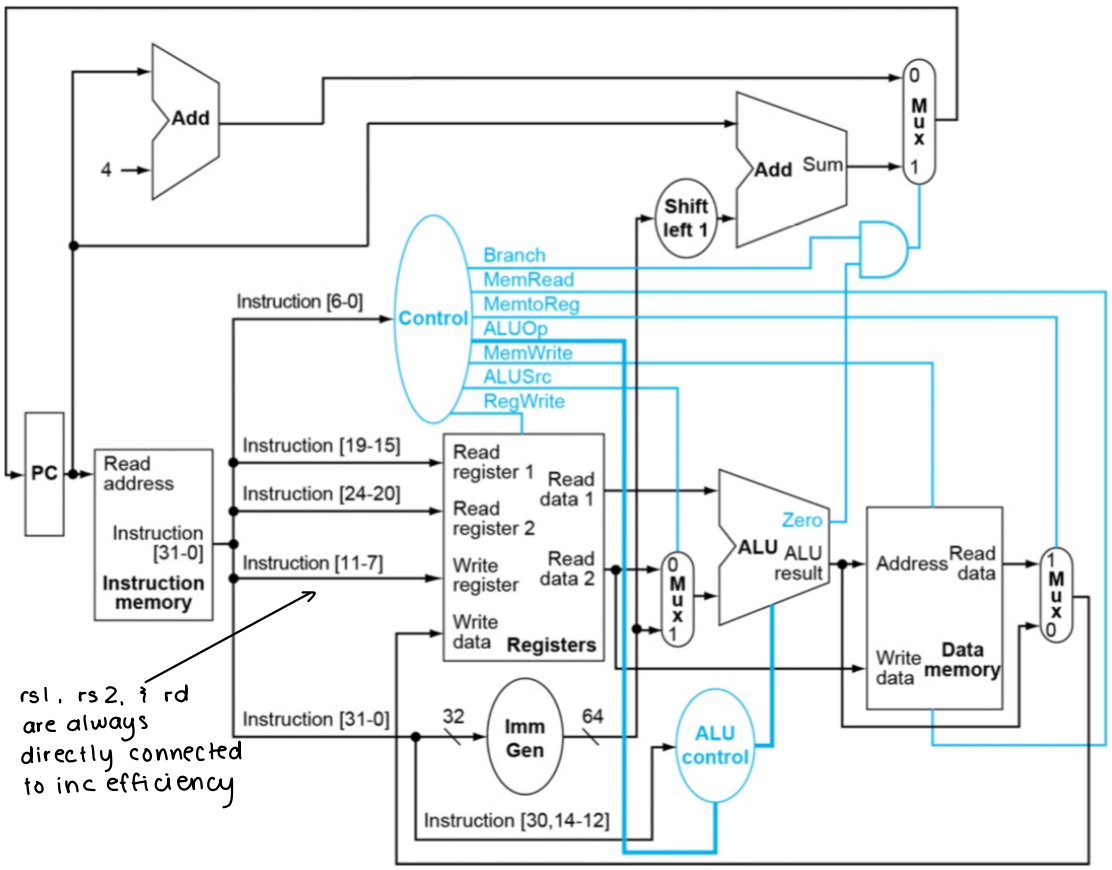
ai	bi	opn	alu func	details
0	0	0 0	a & b	and
0	0	0 1	a | b	or
0	0	1 0	a + b	add
0	1	1 0	a + (-b)	sub
0	1	1 1	a + (-b)	set if less than , result <= "0...0" || msb(a + (-b))

- control signals are set to 0 unless otherwise stated:
  - ↳ R-type: RegWrite = 1, ALUOp = 10
  - ↳ I-type: ALUSrc = MemToReg = RegWrite = MemRead = 1
  - ↳ S-type: ALUSrc = MemWrite = 1
  - ↳ SB-type: Branch = 1, ALUOp = 01

- ALUOp output from control block is input to ALU control block w/ funct field 3 & 2nd msb of funct field 7 (i.e. ins[30])
-

instruction	ALUOp	functn7 instruction[30]	functn3	ALU function	ALU control signals (ai,bi,opn)
ld	0 0	x	xxx	add	0 0 1 0
sd	0 0	x	xxx	add	0 0 1 0
beq	0 1	x	xxx	subtract	0 1 1 0
add	1 0	0	0 0 0	add	0 0 1 0
sub	1 0	1	0 0 0	subtract	0 1 1 0
and	1 0	0	1 1 1	and	0 0 0 0
or	1 0	0	1 1 0	or	0 0 0 1

- final processor core design is shown below:



↳ single-cycle processor implementation (i.e. all ins execute in 1 clock cycle)
↳ load ins is probably critical path (i.e. takes most amount of delay time) b/c it propagates through reg file access, ALU address calc, mem read, & reg file write
↳ performance is poor b/c processor's clock cycle is limited by slowest ins

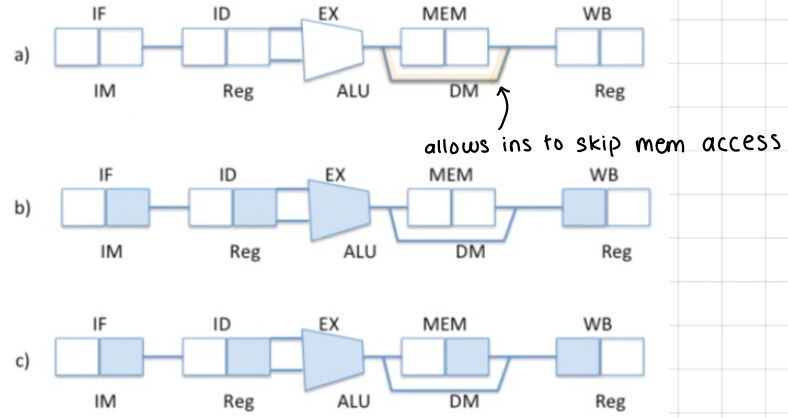
PIPELINING

- pipelining provides method of improving performance
divide processor into 5-stage pipeline (w/ controls that are used in each stage listed):
1) IF: ins fetch
2) ID: ins decode & reg file read
3) EX: ALU execution & address calculation
   - ALUOp
   - ALUSrc
4) MEM: mem access

- Branch
- MemRead
- MemWrite

5) WB · write-back
- MemtoReg
- RegWrite

· processor can be divided into 5 stages as shown below:



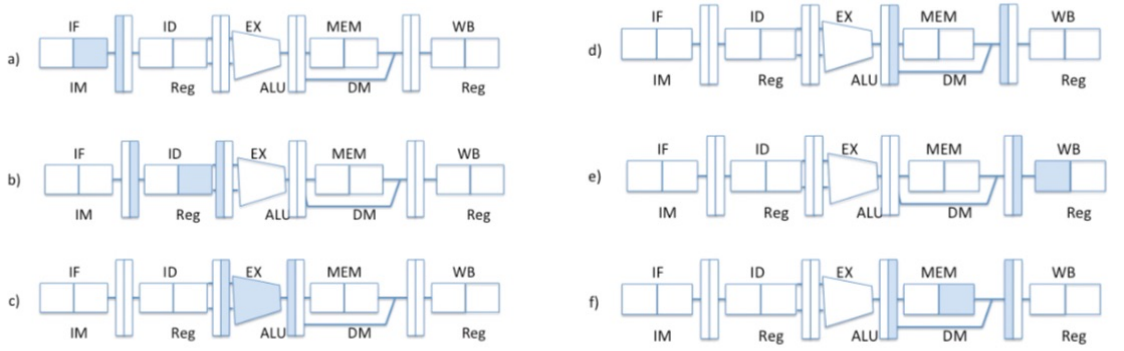
↳ shading indicates what part of flow is used by ins
- on left means a write of a value
- on right means a read of a value

↳ b rep add ins

↳ c rep ld ins
- ALU is used to compute address for data mem where data is read out & then written into reg file in WB stage

· to implement pipelining into hardware, a pipeline register is placed in btwn each of stages so that during 1 clock cycle, all 5 stages may be active performing diff ins

↳



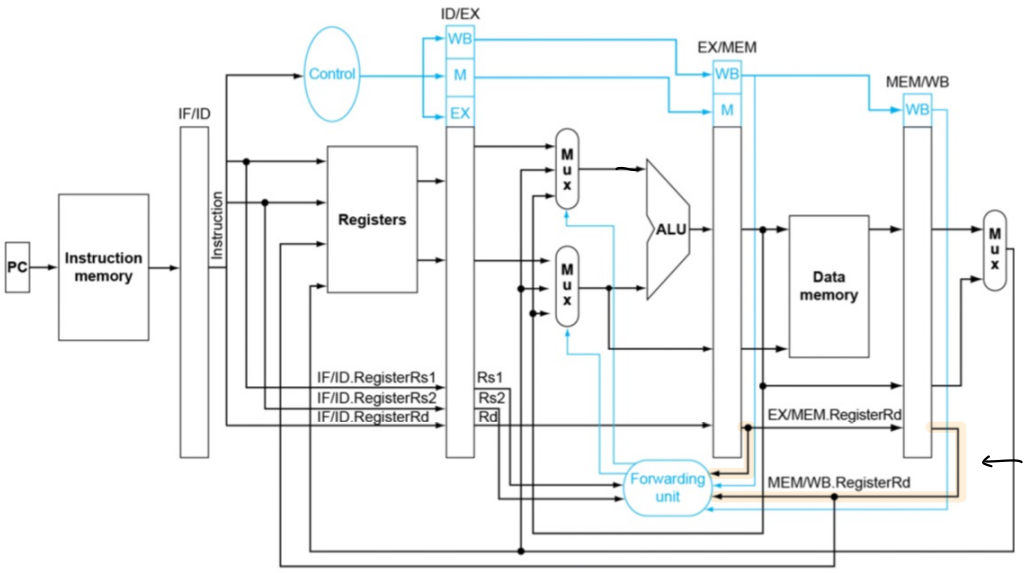
- although ID & WB stages are simultaneously using reg file, ID reads a value for 1 ins & WB writes back a value for another ins during same clock cycle
- a-e rep flow for add ins

· more detailed datapath shown below:

↳ displayed above pipeline reg is forwarding of control signals

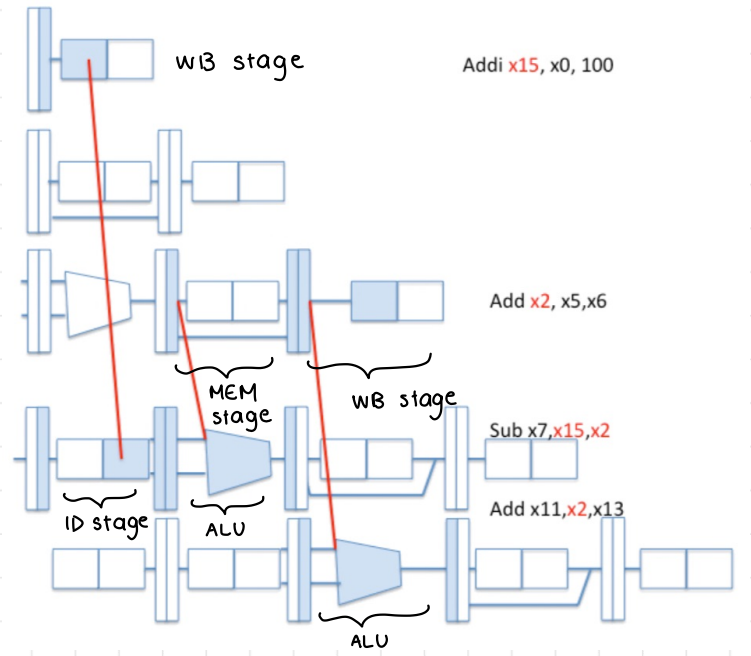
↳ control signals (EX, M, WB) must be placed in appropriate pipe stage

↳ always assume writing to reg occurs before reading from them



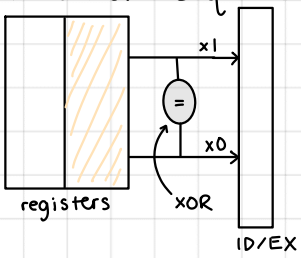
- data hazards occur when ins depends on result of prev ins that hasn't been fully computed yet
  - ↳ e.g. I0: add x1, x2, x3
    I1: addi x4, x1, 1
    I2: add x6, x1, x5
    - reg x1 from I0 isn't written back into reg file until clk cycle 4 but its value needs to be used before then
    - value of x1 is prod in clk cycle 2 by I0 so it can be forwarded to ALU for I1 in clk cycle 3 & I2 in clk cycle 4

e.g. data fwding shown w/ red lines



- not all data hazards can be fixed w/ fwding so we must use a nop (no operation) or stall in ins set
  - ↳ e.g. I0: ld x1, 20(x2)
    I1: add x5, x1, x4
    - x1 isn't available from mem in cycle 3 but it's needed as input to ALU during same clk cycle 3
    - soln is to intro a nop after ld ins
  - ↳ can be avoided if compiler can reorder statements to avoid stall

- control hazards occur w/ branching
  - ↳ e.g. 10: beq x1, x0, addr1
    11: add x5, x1, x4
    ⋮
    addr1:
    - beq output determined by ALU output in cycle 2 ∴ correct addr is fetched in cycle 3
    - if branching, prev 3 ins are turned into nops but this too high of penalty
    - soln is to move beq decision to ID stage ∴ XORing 2 reg values

→ 

EXCEPTIONS AND INTERRUPTS

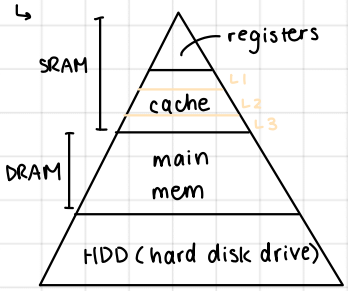
- other than branches, exceptions ∴ interrupts also change control of ins flow execution
  - ↳ exception is unscheduled event that disrupts program execution
  - ↳ interrupt is exception that comes from outside processor
- e.g. external I/O device (interrupt), system reset, invoke OS from user program, using undef ins, ∴ hardware malfunctions
- when exception occurs:
  - ↳ set reg to show cause of exception
  - ↳ address of offending ins is saved
  - ↳ all following ins are flushed
  - ↳ all prior ins are completed

for I/O device request or OS service call:
- ↳ save state of program
- ↳ perform task
- ↳ later, restores program to cont execution
  - may have to complete I/O task execution before resuming program

CHAPTER 5

MEMORY TECHNOLOGIES

- locality is important principle of mem hierarchy design
  - ↳ means that program access small portion of their address space at any 1 time
  - ↳ 2 types:
    - temporal (time): if data unit is accessed, likely to be accessed again soon
    - spatial (space): if data unit is accessed, likely that nearby data will be accessed soon
- mem hierarchy is organized so faster & smaller mem is closest to processor
  - ↳

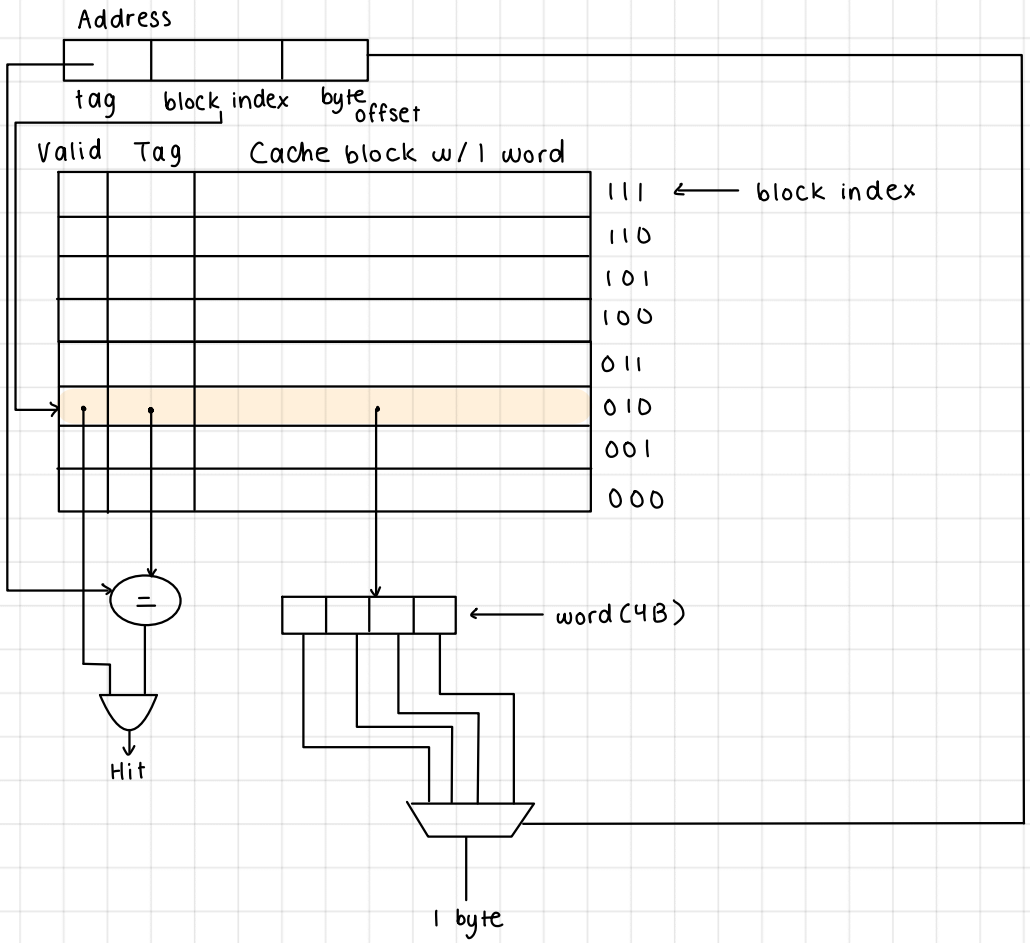


  - ↳ data is only copied btwn 2 adjacent levels at a time
- in 2 level mem hierarchy, there's main mem (lower level) & L1 cache (upper level)
  - ↳ min amount of data is called block / line
  - ↳ hit: data requested by processor is in upper level
    - hit rate is frac of mem accesses in upper lvl
    - hit time is time required to access data in upper lvl (includes time to determine if it's hit/miss)
  - ↳ miss: data requested by processor isn't in upper level
    - miss rate = 1 - hit rate
    - miss penalty is time required to replace block in upper lvl w/ block from lower lvl + time to deliver block to processor
- usually use Harvard architecture w/ single main mem & separate ins cache (i.e. imem) & data cache (i.e. dmem)

CACHES

- in a direct-mapped cache, each main mem address maps to exactly 1 cache block
- e.g. main mem address space uses 7-bit addresses, cache holds 8 blocks, & each block holds 1 word (4B)
  - ↳ each entry in cache would have layout:

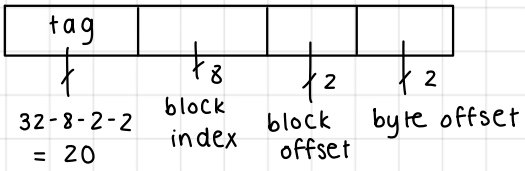
    - valid bit identifies if block is empty
    - since there's 8 blocks & $2^3 = 8$, need 3 bits for block index (identifies blocks)
    - tag is 2 bits to separate addresses in same block (since main mem uses 7-bit addresses)
    - byte offset identifies each byte in 4B word (need 2 bits for 4B word)
- to see if there's a cache hit/miss:



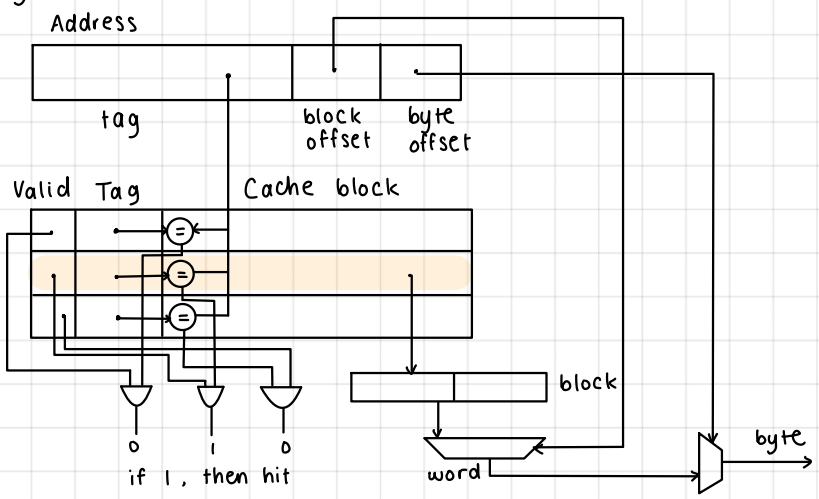
- size of cache accounts for data only
  - ↳ i.e. doesn't include valid bit, tag, etc.
- e.g. find address field sizes for 4KB = 4096B direct-mapped cache for RV32I; assume cache block = 4 words

$$\#\text{ cache blocks} = 4096\,B\left(\frac{1\text{ word}}{4\,B}\right)\left(\frac{1\text{ block}}{4\text{ words}}\right)$$

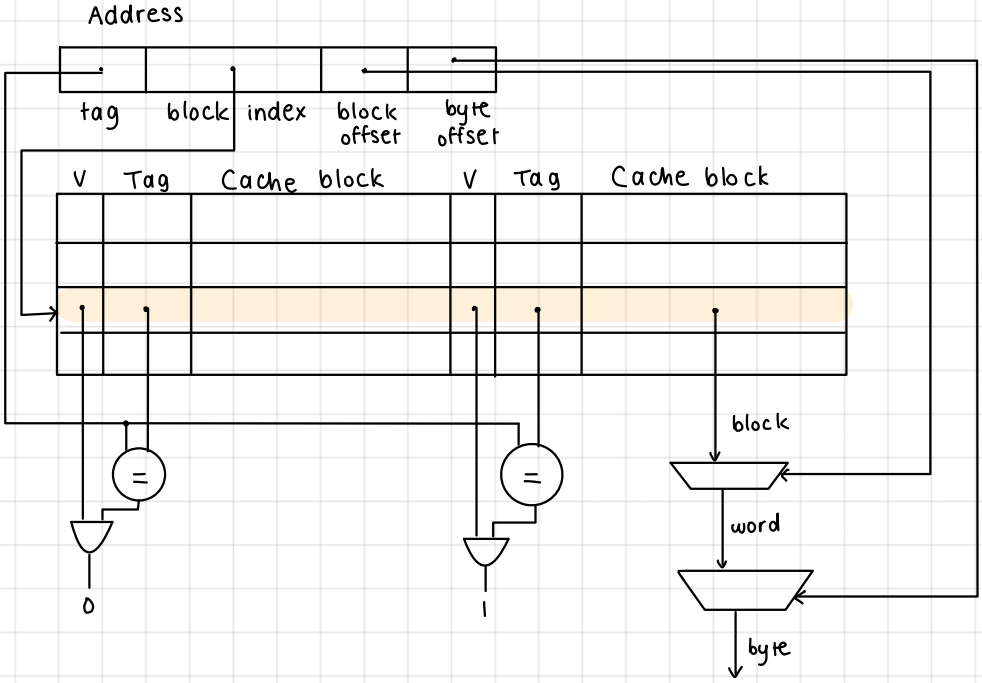
$$= 256\text{ blocks} = 2^8 \longrightarrow \text{8-bit block index}$$



- ↳ block offset identifies specific word in block
  - 4 words / block means 2-bit offset
- in a fully associative cache, block can be placed anywhere in cache instead of forcing each mem address to a specific block index
  - ↳



- n-way **set associative** cache implements combo of direct-mapped & fully associative placement
  - ↳ each set consists of n blocks
  - ↳ block is mapped to set using index & all blocks in set are searched for a match using tag field
  - ↳ e.g. 2-way set associative cache:



- e.g.

Example: consider a 4-way set associative cache with cache size of 4096 blocks, 4 word block size, with byte-addressable memory, and 64-bit address.

  - ↳ $4096/4 = 1024$ sets $= 2^{10}$
  - ↳ 10-bit block index field
  - ↳ 1 word = 4 bytes so 2-bit byte offset
  - ↳ 4 words/block so 2-bit block offset
  - ↳ $64 - 10 - 2 - 2 = 50$-bit tag field
- if there's cache miss, typically use LRU (least recently used) scheme is used to select which block to remove
- 3 types of **cache misses**:
  - ↳ compulsory: cache empty
  - ↳ conflict: cache not full but there's collision
  - ↳ capacity: cache is full
- for **cache read miss**, processor stalls until mem access is complete:
  - ↳ send original pc to mem
  - ↳ instruct mem to do read of correct data
  - ↳ write cache entry
  - ↳ restart ins execution & it'll refetch ins
- for ins caches, early restart scheme is employed
  - ↳ i.e. execution resumes as soon as requested word of block is returned & don't wait for transfer of full block
- for data caches, word requests are less predictive so processor must stall
  - ↳ another scheme is to transfer requested word first then resume execution while transferring remaining words in parallel
- when ins writes to data cache, main mem & data cache will be inconsistent

- to write to mem, we use **write buffer** that'll write in parallel to main mem w/o stalling processor
- 2 schemes to write to main mem:
  - ↳ **write-through**: write to write buffer immediately
  - ↳ **write-back**: write later when block is replaced
    - need dirty bit to indicate block was modified
- to deal w/ write miss, can either **write allocate** (i.e. bring block into cache) or **no write allocate** (i.e. write directly to write buffer)
- some important **equations**:
  - ↳ CPU time = (CPU execution cycles + mem-stall cycles) * clock period
  - ↳ mem-stall cycles = read-stall cycles + write-stall cycles
    - read-stall cycles = $\frac{\text{reads}}{\text{program}}$ * read miss rate * read miss penalty
    - write-stall cycles = $\frac{\text{writes}}{\text{program}}$ * write miss rate * write miss penalty + write buffer stalls
  - ↳ if write buffer stalls are negligible:
    - mem-stall cycles = $\frac{\text{mem accesses}}{\text{program}}$ * miss rate * miss penalty
      $= \frac{\text{ins}}{\text{program}} * \frac{\text{misses}}{\text{ins}} *$ miss penalty
    - avg mem access time = time for hit + miss rate * miss penalty
      = cache access time (including hit detection) + miss rate * miss penalty

VIRTUAL MEMORY

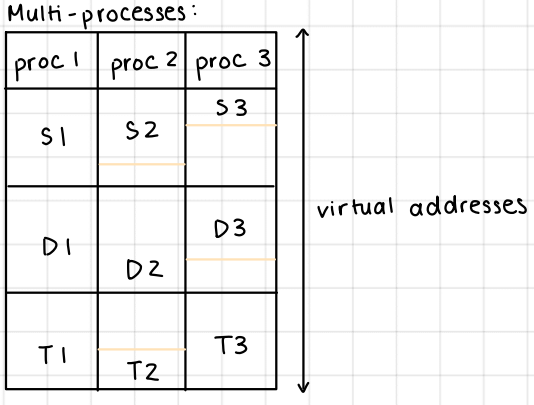
- **virtual mem** is lvl of mem hierarchy that manages caching btwn main mem & disk / secondary mem

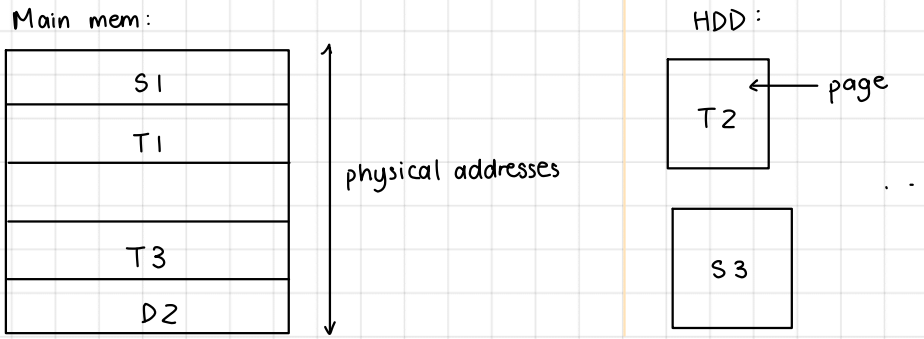
comparison btwn cache mem vs virtual mem:

Cache processor – cache – main mem	Virtual Mem main mem – HDD
↳ cache blocks (fixed size)	↳ page (fixed size)
↳ cache to main mem	↳ main mem to HDD / flash
↳ cache miss	↳ page fault
↳ block replacement	↳ page replacement
• LRU, LIFO, etc.	• LRU
↳ valid bit	↳ valid bit
↳ DM, FA, n-way SA	↳ fully associative
↳ write through / back	↳ write back
↳ physical address	↳ virtual address gets translated to physical address

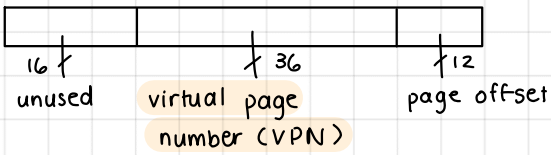
- many **virtual machines** (VMs) share same mem so virtual mem allows for safe sharing among several programs
  - ↳ diff processes can have same virtual addresses but distinct physical addresses:

Multi-processes:

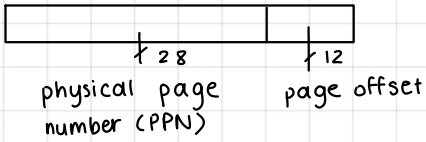




- address translation / mapping is translating virtual to correct physical address
  page offset is determined by page size
- e.g. RV64I uses 64-bit virtual addresses w/ pages of size 4KB = 2^{12} B
  - ↳ virtual address (64-bit):

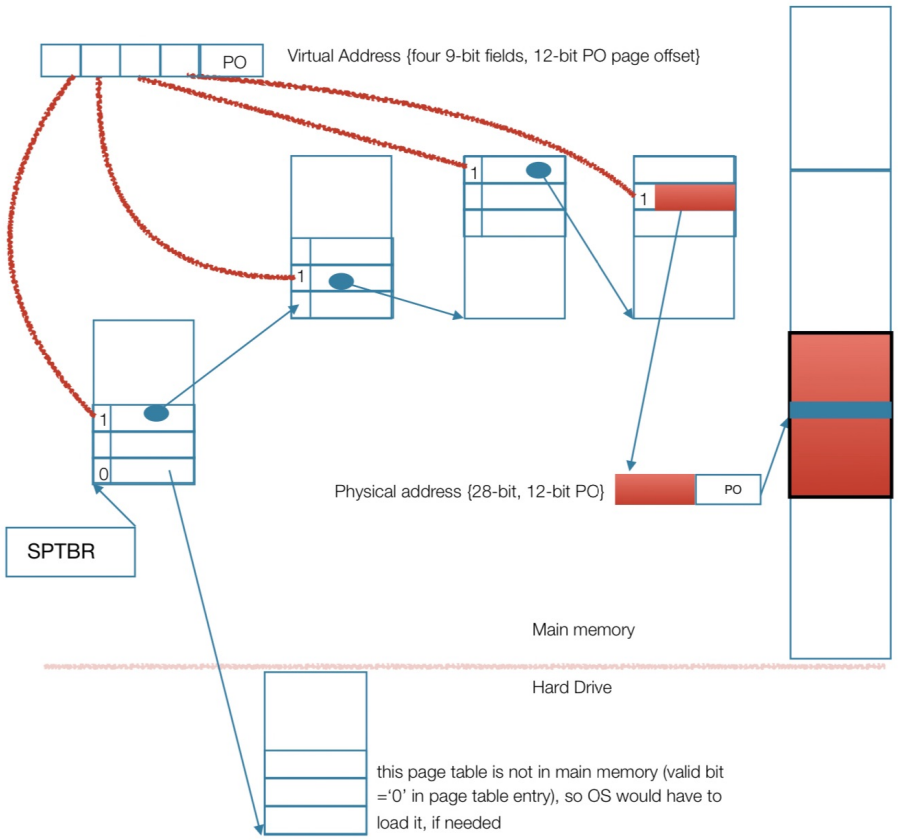


  - ↳ physical address (40-bit):



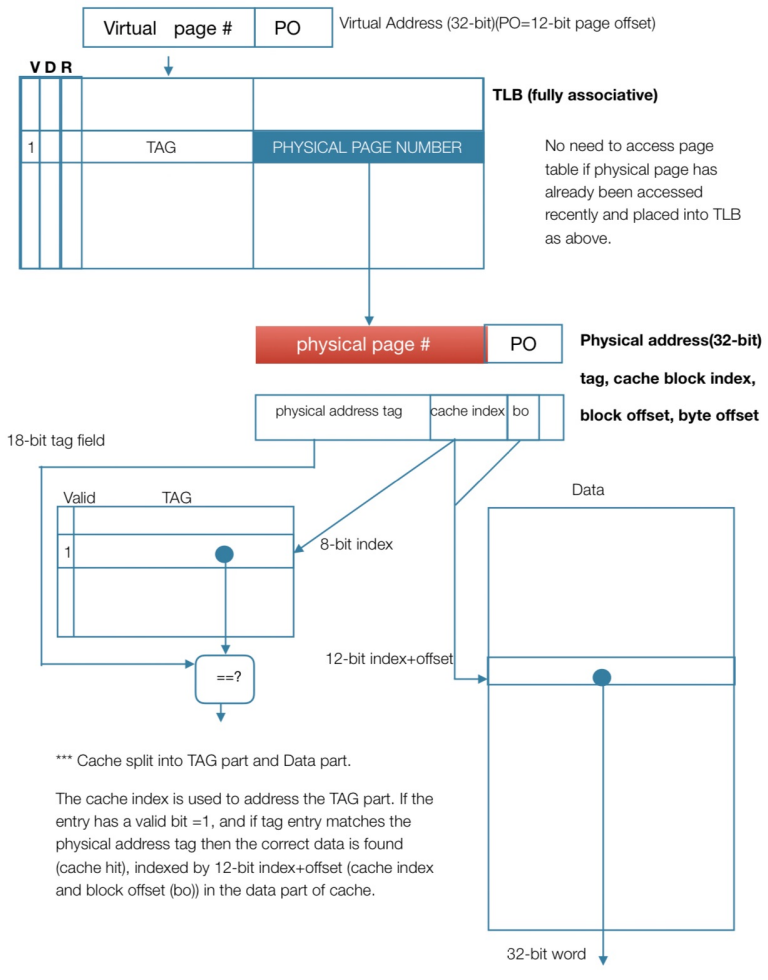
- page faults are handled in software
- each program has page table, which maps VPN to PPN
  - ↳ page table may contain entries for pages not currently in mem
  - ↳ page table register (PTR) holds address / location for page table in mem
  - ↳ fully associative so no tags required
    - table contains mapping for every possible virtual page
- state of VM is specified by PTR, program counter (PC), & registers
  - ↳ forms a process
- if valid bit 0, page fault occurs
  - ↳ OS is given control & has to find page in next level
- OS creates swap space on flash / disk for all pgs in process
- in RISC-V, multi-level page table is used to support large page sizes
  - ↳ 4 levels used
  - ↳ each page table is indexed w/ 9 bits of virtual address
  - ↳ each pg table entry is 8B so each table is $2^9 \cdot 2^3 = 4KB$
    (entry indexes; bits per entry)
- translation for VPN is slow so use translation-look aside buffer (TLB), which is special cache to keep track of recent translations (fully associative)
  - ↳ TLB is accessed before pg table on every reference
- if TLB miss occurs, load translation from last lvl pg table
- typical TLB values:
  - ↳ TLB size: 16-512 entries
  - ↳ block size: 1-2 pg table entries
    - 4B-8B each
  - ↳ hit time: 0.5-1 clock cycle
  - ↳ miss penalty: 10-100 clock cycles
  - ↳ miss rate: 0.01% - 1%

4-LEVEL PAGE TABLE ADDRESS TRANSLATION EXAMPLE



The OS loads the page tables for each process, by loading the process's page table register value into SPTBR (supervisor page table base register), which holds the starting address of the first page table.

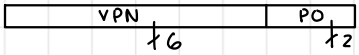
TRANSLATION-LOOKASIDE BUFFER (TLB) EXAMPLE



Toy Example: Given 8bit virtual address space, main memory of 64B, page size of 4B, and page table entry size of 1B,
Find the size of VPN, PPN, PO. How many levels of multi-level page table are needed?

↳ pg size = 2^2 B → PO : 2-bit

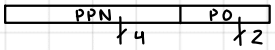
↳ VA



- VPN : 6-bit

↳ main mem size = 2^6 B → # pages in MM = $\frac{2^6 \text{ B}}{2^2 \text{ B/pg}} = 2^4$

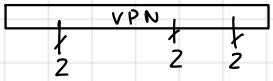
- PA



- PPN : 4-bit

↳ $\frac{\text{PT size}}{\text{PT entry size}}$ = # entries

$\frac{2^2}{2^0} = 2^2$ → 2-bit PTE index



- need 3-lvl pg table

· data can't be present in cache unless it's also in main mem

· when OS writes page to disk, must flush contents of page from cache

↳ also modifies TLBs & page tables so page fault is generated if page is requested

· process / context switch is when OS changes from running process P1 to P2

↳ pg table reg changes to point to P2's pg table

↳ TLB is cleared

↳ costly in time so alternative is to extend virtual address space by adding process / task identifier