Instruction set architecture and implementation

Instruction set architecture

Central processing unit (CPU)

- The CPU consists of three main parts:
  - ↳ 1) Control unit: Fetch instructions from memory and determines their type. Then coordinates data flow around CPU as instruction is executed.
  - ↳ 2) Arithmetic logic unit: Performs simple arithmetic and logic functions on data within the CPU
  - ↳ 3) Registers: Small, high speed memory used to store temp. results and control info.
- Instructions are executed in a fetch-decode-execute cycle:
  - ↳ Fetch instruction from the MM address pointed to by the program counter (PC), then increment the PC
  - ↳ Decode the instruction. If the instruction uses data from memory, calculate the relevant addresses and fetch data from memory.
  - ↳ Execute the instruction and store the results.
- The CPU's instruction set is a low level language. We use a compiler to translate high level languages (C++) into instructions CPUs can understand.
- CPU instruction sets can be large (complex instruction set computer CISC) or small (reduced instruction set computer RISC)

Measuring performance

- The only really meaningful measure of performance is the time the computer takes to execute a desired task. (total time = CPU crunching + system functions + other processes).
  UNIX time command: real = user + sys + real − user − sys
- The execution time is the time to perform a single job. The throughput is the total work done per unit time.
  An extra CPU may increase throughput but may not decrease execution time.
- The runtime of a task may be factorised as follows:

$$\text{run-time} = \text{no. instructions} \times \frac{\text{no. clock cycles}}{\text{instruction}} \times \frac{\text{no. seconds}}{\text{clock cycle}}$$

→ There are three main ways to improve performance for a fixed instruction set:
- ↳ 1) Reduce instruction count for the program count — compiler design.
- ↳ 2) Reduce the avg no. of clock cycles per instruction (CPI) — hardware design.
- ↳ 3) Reduce the clock cycle time — hardware design.

* There is often a trade-off b'wn 2 and 3 (design techniques which reduce the clock cycle time tend to increase the CPI.

Amdahl's law

- Amdahl's law states that the performance enhancement possible w/ a given improvement is limited by the amount the improved feature is used.
- The speed up of a program is given by

$$\text{speedup(program)} = \frac{\text{old time}}{\text{new time}} = \left[(1 - \text{fraction(enhanced)}) + \frac{\text{fraction(enhanced)}}{\text{speedup(enhanced)}} \right]^{-1}$$

e.g.: 4x speed up for instruction used 90% of the time: $\text{speedup} = \left(0.1 + \frac{0.9}{4}\right)^{-1} = 3.077$

10x speed up for instruction used 50% of the time: $\text{speedup} = \left(0.5 + \frac{0.5}{10}\right)^{-1} = 1.818$

→ the former is more effective than the latter.

- The essence of Amdahl's law is to make the common case fast. (which may differ quite significantly → diff. design decisions for diff. applications)

classifying instruction set architectures (ISA).

we typically use the same ISA for decades.

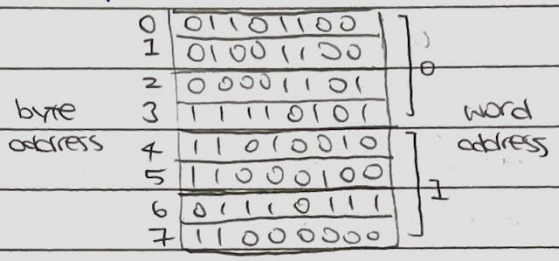
- The ISA defines the instructions, registers, addressing modes and instruction formats of the processor. It is designed to serve a no. of diff. implementations
- ISAs can be classified based on operand storage:
  - ↳ Stack ISA: operands taken from the stack — add — stack(1) ← stack(2) + stack(1)
  - ↳ Accumulator ISA: one operand is the accumulator — lda 0xE000 — accumulator ← word @ 0xE000
  - ↳ General purpose register ISA: all operands declared — lw $8, 0($2) — reg#8 ← word at address in reg#2
- * Instruction = opcode + operand
  - ↳ opcode: type of operation to perform (eg. add, sub, and)
  - ↳ operand: data to be operated on (registers, memory words)

classifying GPR architectures.

- GPR architectures are the most dominant:
  - ↳ Most general for code generation
  - ↳ Lots of temp. variables → ↓ memory traffic
  - ↳ well-suited for high-level languages
  - ↳ All operands named → longer instructions
- The key parameters of GPR architectures are:
  - ↳ no. of operands per ALU instruction: 2 (VAX), or 3 (MIPS) ← one operand is both the source and destination
  - ↳ max. no. of memory references: can vary from 0 (MIPS) to 3 (VAX).
- A load-store architecture (max. no. of memory references = 0 → only load/store instructions can access memory) w/ 3-operand format results in simple, fixed-length instructions and simpler code generation.
- e.g.: VAX: add $1,$2 — reg#1 ← reg#1 + reg#2 — add $1,0xE000 — reg#1 ← reg#1 + word @ 0xE000
  MIPS: add $1,$2,$1 — lw $2, E000($0) — reg#2 ← word @ 0xE000; add $1,$2,$1 — reg#1 ← reg#1 + reg#2

Memory addressing

- Memory consists of an array of locations each w/ an address.
- Usually, each address stores an 8-bit quantity — memory is byte addressable.
  A machine w/ 32-bit address (i.e. 2^{32} memory locations) can store $2^{32} \times 8$ bits = 4GiB
- Memory is arranged in words, corresponding to the units of data manipulated by the CPU (i.e. the size of the registers)
- A 32-bit processor (eg. MIPS) uses 32-bit words, so we have 4 bytes per word. To read consecutive words from memory, the (byte) address is incremented by 4 each time.



* Although memory is byte addressable, word addresses are useful (for analysing caches).

$$\text{word address} = \left\lfloor \frac{\text{byte address}}{4} \right\rfloor$$

- Note there are two conventions for byte ordering within words.
  ↳ e.g. Consider storing the word 0x12457802 in memory location 0

Big endian					Little endian				
Address	0	1	2	3	Address	3	2	1	0
Contents	12	45	78	02	Contents	12	45	78	02

This diff. is transparent to the programmer but it can be a pain when a little endian computer communicates w/ a big endian computer.

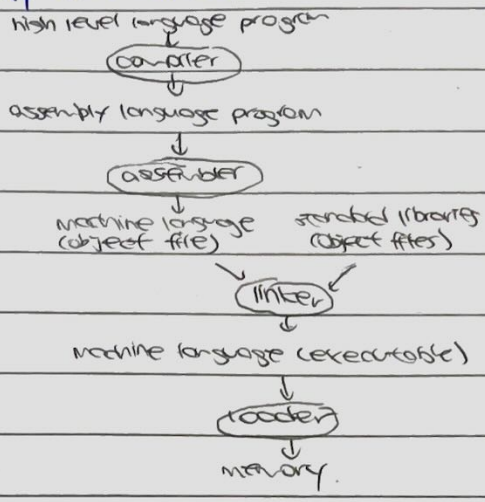
- Memories occasionally make errors → most memory systems use error detecting/correction codes, w/ extra bits to provide the necessary redundancy.

MIPS ISA

- MIPS is a load-store architecture w/ fixed-size 32-bit instructions.
- There is a core instruction set and 32 general purpose registers; there are also special instructions and registers for the system coprocessor and floating point coprocessor.
- MIPS operands can be registers or memory words:
  ↳ 32 registers ($0, $1 .. $31): very high speed, data must be in registers for ALU operations
  ↳ 2^{30} memory words (mem[0], mem[1] .. mem[$2^{30}-1$]): accessed only by load/store instructions.
* MIPS has 2^{32} memory locations, w/ each memory location holding 1 byte. Each word is 4 bytes, so sequential words differ by 4 byte addresses.
- The instruction set can be divided into groups:
  ↳ Computational: ALU operations (on reg. val)
  ↳ Jump and branch: change flow of control
  ↳ Load/store: Move data btwn reg/memory
  ↳ Coprocessor: Coprocessor operations.

Translation hierarchy

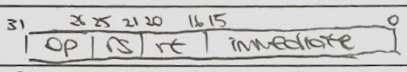
- The translation hierarchy is as follows:



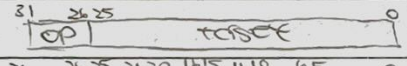
- The symbolic representation of the instructions is the assembly language of the computer.
  The numeric representation of the instructions is the machine language program ← loaded and executed in memory
- The assembler does more than a simple mapping from symbols to no. It also allows symbolic pseudo instructions (eg: move is implemented using add under the hood).

MIPS instruction formats.

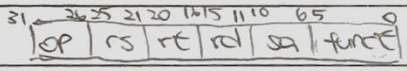
- There are three instruction formats in MIPS.
  - ↳ I-type (immediate):



  - ↳ J-type (jump):



  - ↳ R-type (register):



- The diff. fields are:
  - ↳ op (6-bit): operation code
  - ↳ rs (5-bit): source register specifier
  - ↳ rt (5-bit): target register / branch condition
  - ↳ immediate (16-bit): immediate
  - ↳ target (26-bit): jump target address
  - ↳ rd (5-bit): destination register specifier
  - ↳ shamt (5-bit): shift amount
  - ↳ funct (6-bit): function field.

Addressing modes.

- The addressing mode describes how an instruction accesses its operands.
- Some popular addressing modes are:
  - ↳ Register: operand is a register
  - ↳ Immediate: operand is a const. within the instruction
  - ↳ displacement (base): operand in memory, address = register + offset.
  - ↳ PC-relative: address = PC + const. (for branches).
  - ↳ Direct: operand in memory, address is a const. in the instruction
  - ↳ Indirect: operand in memory, address in register

available in MIPS.

Zero-extend and sign-extend

- Often it is necessary to work on no. smaller than 32 bits. We can do operations on n-bit/32-bit numbers by extending the n-bit number to 32-bits.:
  - ↳ zero-extend: Fill the upper bits w/ zeros
  - ↳ sign-extend: Fill the upper bits w/ the sign bit (ie. +ve no: 0 ; -ve no.: 1).
- e.g.: extend -5 (1011) (i) zero extend: 0000 0000 0000 0000 0000 0000 0000 1011
  (ii) sign extend: 1111 1111 1111 1111 1111 1111 1111 1011

Special registers.

- MIPS uses the following special registers:
  - ↳ $0: Always set to zero
  - ↳ $1, $26, $27: Reserved for assembler / OS.
  - ↳ $4-$7: First 4 arguments to procedures
  - ↳ $31: Return address for last procedure call
  - ↳ $29: stack pointer

Addition and subtraction instructions

- The addition and subtraction instructions are as follows:
  - ↳ addition: `add rd, rs, rt` reg rd ← reg rs + reg rt
    `addi rt, rs, immediate` reg rd ← reg rs + immediate
  - ↳ subtraction: `sub rd, rs, rt` reg rd ← reg rs - reg rt.
- The move pseudo-instruction (accepted by assembler but not ISA) uses the add ISA instruction.
  - ↳ eg: `move $8, $18` ⇔ `add $8, $0, $18` reg #8 ← reg #18.

Load and store instructions

- The load word and store word instructions are as follows:
  - ↳ load word: `lw rt, offset(base)` reg rt ← mem[base address + offset]
  - ↳ store word: `sw rt, offset(base)` reg mem[base address + offset] ← rt
- In general, two instructions are req. to load a 32-bit const. into a register. We first load the upper 16 bits using LUI, then load the lower 16 bits using ORI
  - ↳ eg. reg#15 ← 0x07FFFFFF: `lui $15, 0x07FF`
    `ori $15, $15, 0xFFFF`
- * If the upper 17 bits are all 1, we can simply use addi (which sign extends the 16-bit const)
  - ↳ eg. reg#15 ← 0xFFFFFF00: `addi $15, $0, 0xFF00`

Shift Instructions

- shift instructions allow register contents to be shifted by up to 31 bits. the shift amount is either specified in shamt field (SLL, SRL, SRA) or lower 5 bits of a register (SLLV, SRLV, SRAV)
- for shift right, we can either zero extend (shift right logical) or sign extend (shift right arithmetic)
- the shift instructions are as follows:

↳ shift left:	sll	rd, rt, sa	reg rd ← shiftleft(rt, sa) + zero pad
	sllv	rd, rt, rs	reg rd ← shiftleft(rt, rs[5:]) + zero pad
↳ shift right:	srl	rd, rt, sa	reg rd ← shiftright(rt, sa) + zero extend
	srlv	rd, rt, rs	reg rd ← shiftright(rt, rs[5:]) + zero extend
	sra	rd, rt, sa	reg rd ← shiftright(rt, sa) + sign extend
	srav	rd, rt, rs	reg rd ← shiftright(rt, rs[5:]) + sign extend

- we can use shift left/right to multiply/divide by powers of 2.
  - ↳ eg. reg #5 ← reg #5 × 2^4 sll $5, $5, 4
    reg #4 ← reg #4 ÷ 2^5 sra $4, $4, 5

Multiply and divide instructions.

- Multiply and divide instructions use two special 32-bit registers HI and LO.
  - ↳ Multiplication: HI: upper 32 bits of product; LO: lower 32 bits of product.
  - ↳ Division: HI: 32-bit remainder; LO: 32-bit quotient.
- The instructions MFHI and MFLO move values from HI and LO to other registers
- The multiply and divide instructions are as follows:

↳ Multiply	mult	rs, rt	HI, LO ← reg rs × reg rt
↳ Divide	div	rs, rt	HI, LO ← reg rs ÷ reg rt

(Then we move the results from HI and LO mfhi rd / mflo rd)

Conditional branching.

- The conditional branching instructions are as follows:
  - ↳ Branch if equal: `beq rs,rt,offset` — increment PC by 4+4×offset if reg rs = reg rt
  - ↳ Branch if not equal: `bne rs,rt,offset` — increment PC by 4+4×offset if reg rs ≠ reg rt.
- The 16-bit offset is interpreted as a word offset (not byte offset), so we increment the PC by the usual 4 + 4×offset. (PC-relative addressing)

* The assembler relieves the programmer from calculating offset addresses – we just use labels.

- The MIPS ISA does not have a full set of branch instructions, so we must additionally use the SLT/SLTI instruction to implement them. `slt rd,rs,rt` — if (reg rs < reg rt), reg rd = 1 else 0; `slti rt,rs,immediate` — if (reg rs < immediate), reg rd = 1, else 0
  - ↳ Branch if less than or equal: `ble $a,$b,dest` ↪ `slt $1,$b,$a, beq $1,$0,dest`
  - ↳ Branch if less than: `blt $a,$b,dest` ↪ `slt $1,$a,$b, bne $1,$0,dest`
  - ↳ Branch if greater than or equal: `bge $a,$b,dest` ↪ `slt $1,$a,$b, beq $1,$0,dest`
  - ↳ Branch if greater than: `bgt $a,$b,dest` ↪ `slt $1,$b,$a, bne $1,$0,dest`.

* When the assembler translates the pseudoinstructions, the reserved reg. $1 is used for temp. storage

Unconditional branching.

- Jump instructions enable unconditional branching. The jump destination is either specified in the instruction (J, JAL) or the address stored in the register (JR, JALR)

* The jump and link (JAL) instruction stores the incremented PC in register $31, so it is possible to return after a procedure → the return instruction is simply `jr $31`.

- The unconditional branching instructions are as follows:
  - ↳ Jump: `j target` — jump to address calculated/target
    `jr rs` — Jump to address in reg rs
  - ↳ Jump and link: `jal target` — jump to address calc w/ target, store incremented PC in $31.
    `jalr rs,rd` — Jump to address in reg rs, store incremented PC in reg rd

* The 26-bit target is interpreted as a word address (not byte address) → we shift left twice to get the 28-bit byte address, we then replace the lower 28 bits of the PC w/ this address.

→ The linker/loader must be careful to avoid placing a program across a $2^{28} \times 8$ bit = 256 MiB boundary, otherwise, we need to use the JR, JALR instructions instead.

↳ eg: target field = 0000...1000, PC = 0001.... → 0001 0000....1000 00 (×4 for byte address)

Procedures

- We can call a procedure using the JAL / JALR instructions
- If a procedure calls another procedure, the return address must be stored on the stack. (The MIPS stack fills from the top down. By convention, the address of the bottom of the stack is stored in register $29.)
- Procedure calls also involve passing parameters. By convention, the first 4 are put in registers $4–$7, and the rest on the stack.
- If another procedure is called / a procedure req. the use of registers as scratch space, we use one of two conventions for saving + restoring registers across calls:
  - ↳ caller save: the calling routine saves the registers on the stack
  - ↳ callee save: the called routine saves the registers on the stack.

* In general, the structure for using the stack is as follows:

```
addi $29, $29, -4n      adjust stack pointer
sw reg1, 0($29)
sw reg2, 4($29)         save reg1...regn on stack for scratch use.
sw regn, 4n($29)
(procedure main body)
lw reg1, 0($29)
lw reg2, 4($29)         restore reg1...regn from stack
lw regn, 4n($29)
addi $29, $29, 8        adjust stack pointer
jr $31                  return from procedure
```

(The above code uses callee save. Move || to calling procedure for caller save)

Traps, exceptions and interrupts.

- Certain instructions will trap on "2's-complement overflow". This means the CPU will trigger an exception or interrupt if an overflow occurs.
- This is an unplanned procedure call where the address of the offending instruction is saved in a register and the computer jumps to a predefined address to invoke the appropriate exception handling routine.

RISC and CISC.

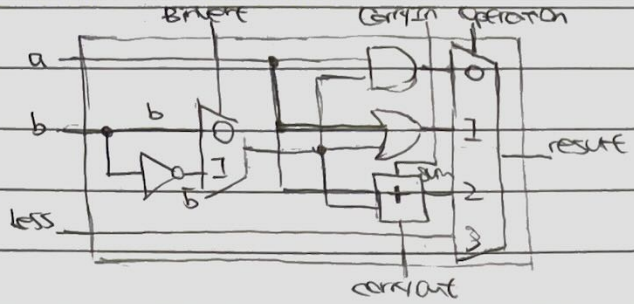
- MIPS is a RISC processor w/ an ISA designed to make the common case fast.

RISC	CISC
- load-store architecture - fixed-length instructions - simple addressing modes - core set of simple instructions - large set of general purpose registers ✓: one instruction per datapath cycle ✓: no need microcode for complex instructions ✗: restricted instruction set → difficult compiler design	- complex instructions taking multiple cycles - any instruction can reference memory - microcode used to control datapath - variable format / length instructions - smaller register set ✓: more compact instructions → lower memory traffic for instructions ✗: fewer registers → more memory traffic for data.

Arithmetic and ALU design

1-bit ALU

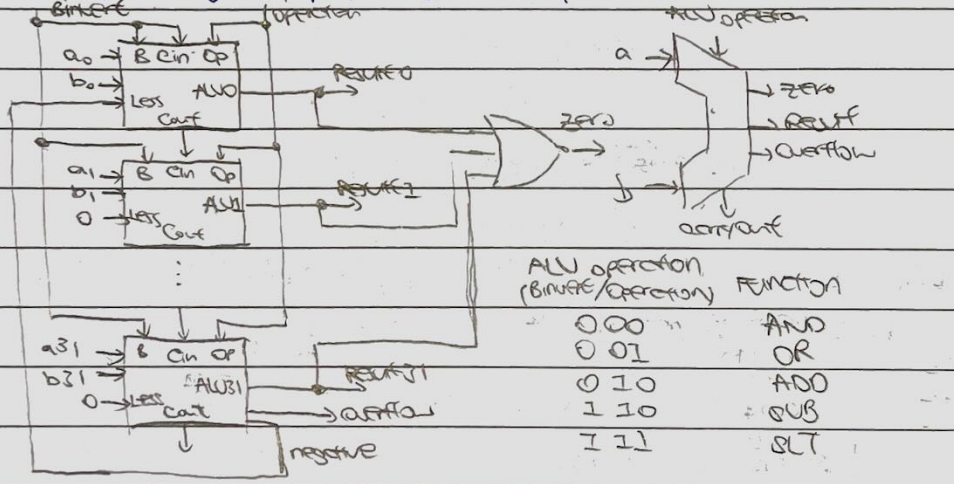
- the ALU performs arithmetic (ADD, SUB) and logical (AND, OR) operations.
- the circuit for a 1-bit ALU is as follows:



- the operation input to the MUX chooses btwn AND, OR, ADD, SUB, SLT functions.
- A full adder is used for ADD (and SUB, SLT) functions, and has 3 inputs and 2 outputs:
  - ↳ $CarryOut = a \cdot CarryIn + b \cdot CarryIn + a \cdot b$
  - ↳ $Sum = a \cdot \bar{b} \cdot \overline{CarryIn} + \bar{a} \cdot b \cdot \overline{CarryIn} + \bar{a} \cdot \bar{b} \cdot CarryIn + a \cdot b \cdot CarryIn$
- For SUB, SLT functions, we set Binvert and CarryIn for the LSB. The SLT function also uses the Less input (see 32-bit ALU implementation)

32-bit ALU

- 32 1-bit ALUs can be connected together, propagating the carry from one bit to the next.



ALU operation (Binvert/Operation)	FUNCTION
0 00	AND
0 01	OR
0 10	ADD
1 10	SUB
1 11	SLT

- SLT can be performed by subtracting b from a, then passing the Less signals to the Result lines (all will be 0 except Result0, which will be 1 if the subtract result is negative).
- the zero line is useful for conditional branching instructions (by calculating a-b).
- ALU31 has some extra logic for overflow detection
- This 32-bit ALU is functional but too slow — it uses ripple carry so it takes 32 additions to produce the result. (A 32-bit adder from truth table is too complex in practice).

Carry lookahead adders

- The main idea of carry lookahead adders is to anticipate the CarryIn signals w/o waiting for them to ripple through the adder.
- The CarryIn of the $(i+1)$-th bit is given by

$$c(i+1) = a_i \cdot b_i + a_i \cdot c_i + b_i \cdot c_i = a_i \cdot b_i + (a_i + b_i) \cdot c_i$$

We can recursively substitute the expression for $c_1, c_2 \ldots c_{(n-1)}$ to obtain an expression for c_n in terms of a, b and c_0 but the no. of terms grows exponentially

* Note that gates w/ a large fan-in (no. of inputs) are impractical (try to limit to 5 i/ps)

- Define the generate g_i and propagate p_i signals as follows:

$$g_i = a_i \cdot b_i \qquad p_i = a_i + b_i$$

g_i is true if bit i of the adder generates a CarryOut indep. of CarryIn

p_i is true if bit i of the adder propagates a CarryIn to a CarryOut.

- We can design a simple, fast 4-bit adder using generate and propagate signals:

$$c_1 = g_0 + p_0 \cdot c_0$$

$$c_2 = g_1 + p_1(g_0 + p_0 \cdot c_0) = g_1 + p_1 g_0 + p_1 p_0 c_0$$

$$c_3 = g_2 + p_2(g_1 + p_1(g_0 + p_0 \cdot c_0)) = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0$$

5 input OR gate

$$c_4 = g_3 + p_3(g_2 + p_2(g_1 + p_1(g_0 + p_0 \cdot c_0))) = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 c_0$$

* Intuitively, c_i is 1 if some adder generates a carry and all intermediate adders propagate a carry.

- We can form a 16-bit adder by connecting four 4-bit adders using a higher level of carry lookahead. — think generate G_i and propagate P_i signals at a 4-bit level.

$$G_0 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 \qquad P_0 = p_3 p_2 p_1 p_0$$

$$G_1 = g_7 + p_7 g_6 + p_7 p_6 g_5 + p_7 p_6 p_5 g_4 \qquad P_1 = p_7 p_6 p_5 p_4$$

$$G_2 = g_{11} + p_{11} g_{10} + p_{11} p_{10} g_9 + p_{11} p_{10} p_9 g_8 \qquad P_2 = p_{11} p_{10} p_9 p_8$$

$$G_3 = g_{15} + p_{15} g_{14} + p_{15} p_{14} g_{13} + p_{15} p_{14} p_{13} g_{12} \qquad P_3 = p_{15} p_{14} p_{13} p_{12}$$

→ the carryIn signals for the 4-bit adders are produced w/o ripple-carry as follows:

$$C_1 = G_0 + P_0 \cdot c_0$$

$$C_2 = G_1 + P_1(G_0 + P_0 \cdot c_0) = G_1 + P_1 G_0 + P_1 P_0 c_0$$

$$C_3 = G_2 + P_2(G_1 + P_1(G_0 + P_0 \cdot c_0)) = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 c_0$$

$$C_4 = G_3 + P_3(G_2 + P_2(G_1 + P_1(G_0 + P_0 c_0))) = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 c_0$$

- The gate delay of a $n = 16$-bit adder w/ $m = 4$ carry-in signals of each level are as follows:

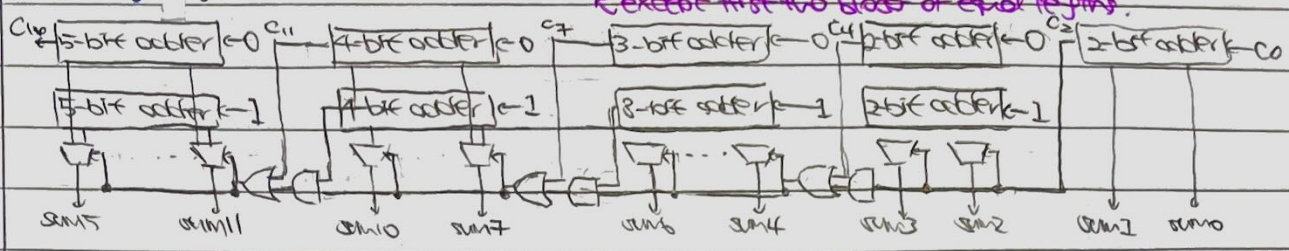
Gate delay	signals available	generated from
0	a_i, b_i, c_0	—
1	g_i, p_i	a_i, b_i
3	G_i, P_i	g_i, p_i
5	C_i	G_i, P_i, c_0
7	c_i	g_i, p_i, C_i
9	sums	a_i, b_i, c_i

ripple carry req. 16×2 gate delays

* P_i, g_i, p_i are AND/OR expressions → 1 gate delay; others are sum of products → 2 gate delays.

Carry select adder

- The principle of operation of a carry select adder is that the hardware for each block is duplicated and two additions are performed in parallel. (one assuming the carry-in to the block is 0 and the other assuming the carry-in is 1).
- After the result of the previous block is known, the correct sum is selected using the MUX.
- Each block uses ripple carry internally. For optimal operation, most blocks should be one bit larger than their immediate predecessor. An optimal design for the 16-bit adder is as follows. ~~Except first two blocks of equal lengths.~~



- Each block should complete its summing just as its MUX select signal (C_2, C_4, C_7, C_{11}) becomes available
  - ↳ The first 2-bit block produces C_2 after 2×2 gate delays → select MUX in second 2-bit block.
  - ↳ C_4 will be available after 1×2 gate delays (total of 3×2 gate delays) → select MUX in 3-bit block
  - ↳ C_7 will be available after 1×2 gate delays (total of 4×2 gate delays) → select MUX in 4-bit block.
  - ↳ C_{11} will be available after 1×2 gate delays (total of 5×2 gate delays) → select MUX in 5-bit block.

Asymptotic complexity.

- The time and space complexity of diff. adders are as follows:

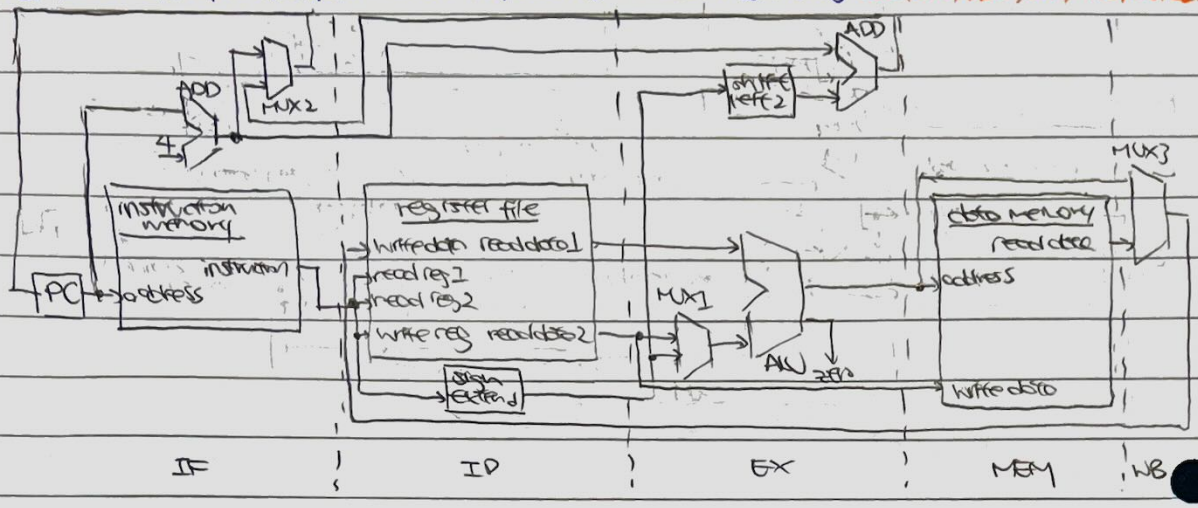
	time	space
ripple carry	$O(n)$	$O(n)$
carry lookahead	$O(\log n)$	$O(n \log n)$
carry select	$O(\sqrt{n})$	$O(n)$

- For a n-bit adder w/ carry-lookahead to predict in carry in $\log n$ levels at each level of the hierarchy, we would need $\log n$ levels and gates w/ a fan-in of $\sim n$.
- Each level takes a const. time to operate → time complexity is $O(\log n)$.
  simple silicon layout gives space complexity $O(n \log n)$.
- For a n-bit carry-select adder, the largest block is of size m, where $1 + \sum_{i=1}^{m} i = 1 + \frac{m(m+1)}{2} \geq n$
  solving for equality gives $m = \frac{1}{2} + \sqrt{\frac{1+8(n-1)}{2}}$. As $n \to \infty$, we have $m \simeq \sqrt{2}\sqrt{n}$.
- The adder takes m time units to operate → time complexity is $O(\sqrt{n})$.
  space complexity is obviously $O(n)$

Datapaths and control

Datapath.

- consider a simple datapath which implements the MIPS commands LW, SW, ADD, SUB, AND, OR, SLT, BEQ.



- For all instructions, we send PC to memory to fetch the next instruction, then read one (LW) or two (others) registers using the instruction fields to select the right registers
- All instructions next use the ALU:
  - ↳ SW, LW need the ALU to calculate an address ↳ BEQ uses the ALU for comparison
  - ↳ ADD, SUB, OR, AND, SLT use ALU for specific execution.
- An arithmetic / logic instruction then writes back the result from the ALU to the register.
  A memory reference instruction accesses memory to read/write a word.
  A branch instruction may need to change the PC depending on the comparison result.
- The registers are stored in the register file (3x 5-bit i/ps for selecting registers to read/write and read/write the selected registers w/ 32-bit I/O).
- An instruction is fetched and executed in a single clock cycle → instructions and data stored in separate caches
- We add MUX to select btwn possible ALU i/ps, "write data" source and extra unit to update the PC after each instruction.

* The regularity of MIPS ISA means we know where to look in an instruction for register no and address offsets.

Datapath control

- To control the datapath, we need to generate automatic control signals for the MUX, ALU, data memory and register file. This is not too complex due to regularity of MIPS ISA
  - ↳ Bits 26-31 define the operation class — decoded by main control unit
  - ↳ For R-format instructions, Bits 0-5 define ALU operation — decoded by ALU control unit.
  - ↳ source/destination registers, branch offset all have defined locations in the instruction fields

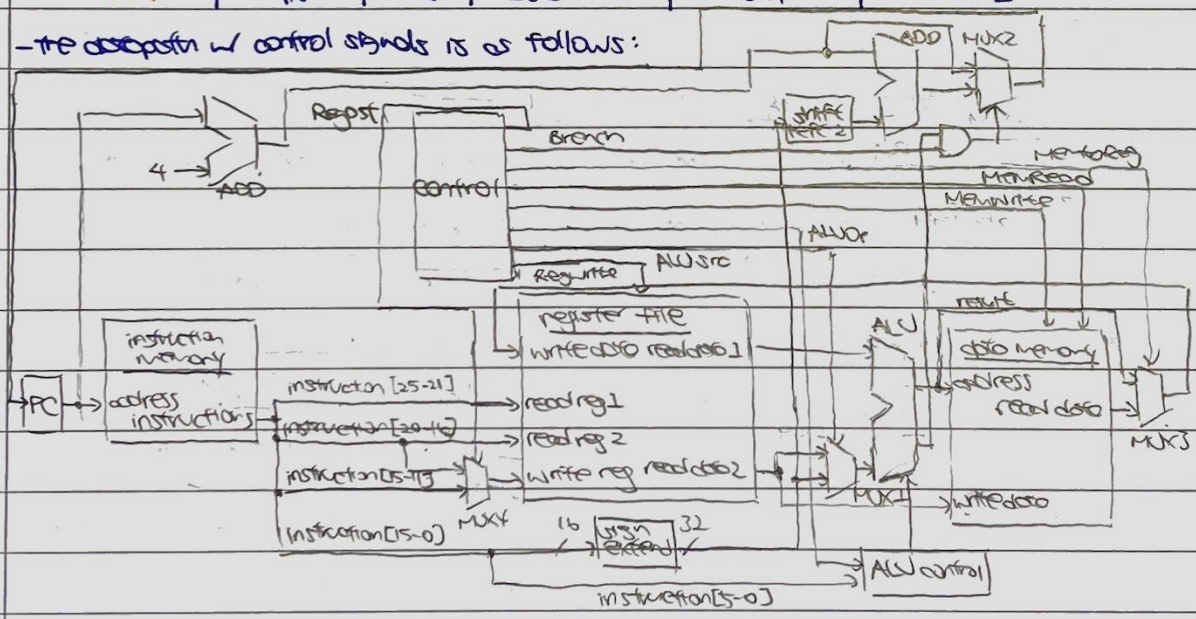
① Main control unit

Signal	Effect when low	Effect when high
MemRead	None	Data at specified address put on read data o/p
MemWrite	None	Data at specified address replaced by write data i/p
ALUSrc	second ALU operand = second regfile o/p	second ALU operand = Bits 0-15 sign extended
RegDst	write reg identified in rt field	write reg identified in rd field
RegWrite	None	Contents of specified reg replaced by write data i/p
Branch	PC ← PC+4	PC ← branch target
MemtoReg	Register write data i/p = ALU	Register write data i/p = data memory

② ALU control unit.

Instruction	Format	ALUOp	Function code (Bits 0-5)	Desired ALU action	ALU control input
LW	I	00	XXX XXX	ADD	010
SW	I	00	XXX XXX	ADD	010
BEQ	I	01	XXX XXX	SUB	110
ADD	R	10	100 000	ADD	010
SUB	R	10	100 010	SUB	110
AND	R	10	100 100	AND	000
OR	R	10	100 101	OR	001
SLT	R	10	101 010	SLT	111

- The datapath w/ control signals is as follows:



- The implementation is single clock cycle → combinational logic for the two control units.

The design is typically implemented using ROM or PLA.

Multiple clock cycle datapath

- The problem w/ a single cycle implementation is that the clock period cannot be shorter than the propagation delay through the longest path in the machine (typically LW) → many instructions could tolerate shorter clock period
- A multiple clock cycle datapath would be faster, w/ diff instructions taking diff no. of cycles. Instructions are broken into steps (IF, ID, EX, MEM, WB) w/ each step taking one cycle.
- The clock can now run faster — only a fraction of each instruction is executed each clock cycle
- A multiple clock cycle datapath needs to maintain state info for control — implemented w/ a state machine (has a set of states, can find next state given current state, give req. control signals)
- For CISC machines, (large no. of types of instructions), the control is too complex to design into a state machine → easier to define complex control using a microprogram.
- Microcode runs on simple, hard-wired computer within the CPU (stored in ROM). This simple computer reads opcodes and generates control signals to drive a big CISC datapath.

Pipelining.

Pipelining.

- Pipelining is a technique which allows multiple instructions to be overlapped in execution. → control to make RISC processors fast. All RISC processors are pipelined
- The instruction execution cycle is split into a no. of pipe stages. The single clock cycle datapath can be pipelined in 5 stages:
  - 1) IF: instruction fetch
  - 2) ID: instruction decode and register fetch
  - 3) EX: execution and effective address calculation
  - 4) MEM: memory access
  - 5) WB: write back to registers
- To complete execution, the instruction uses all pipe stages. However, the next instruction can start once the first instruction has completed the first pipe stage
  → pipelining improves instruction throughput, but not instruction execution time (latency)
- Assume the operation times of the main functional units are:
  - Memory units (Read/write): 10ns
  - ALU and adders: 10ns
  - Register file (Read/write): 5ns.
  - Other units: negligible.
- The time req. for a LW instruction is IF ID EX MEM WB: 10+5+10+10+5 = 40ns

W/o pipelining, 3× LW takes 120ns

W/ pipelining, each pipestage must take equal time → set to 10ns (slowest pipestage) → 3× LW takes 70s

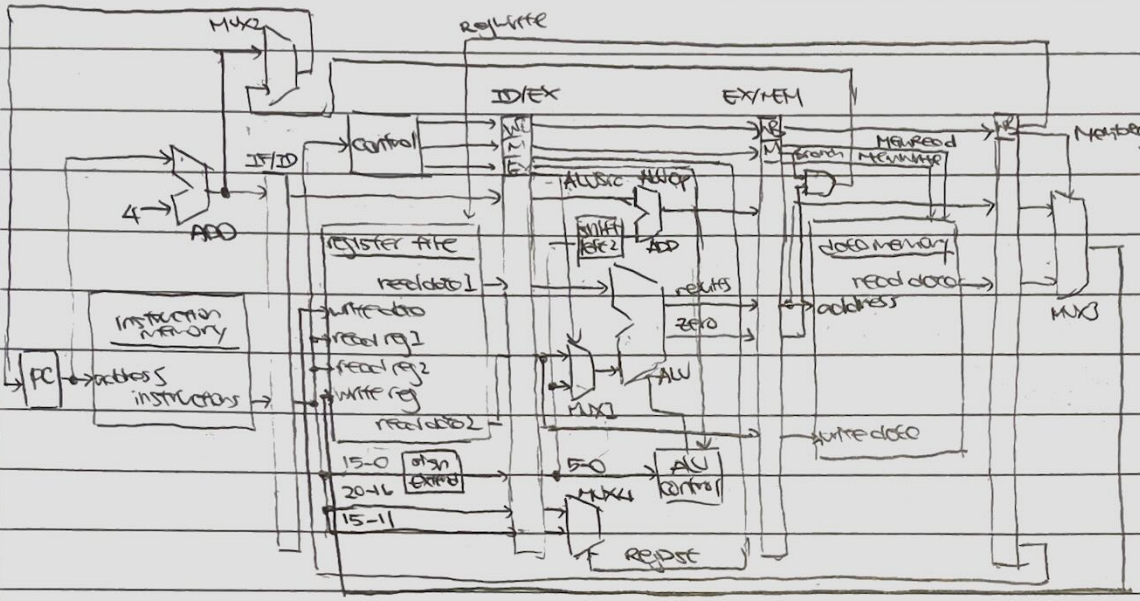
(For a large no. of instructions, we get a 4:1 speedup)

IF	ID	EX	MEM	WB			
	IF	ID	EX	MEM	WB		
		IF	ID	EX	MEM	WB	

- To achieve a good speedup, we req. the pipestages to be balanced.

Implementing the pipeline

- For pipelined implementation, extra pipeline registers must be added to store the result from one pipe stage so it can be used by the next. (a pipeline register btwn each pipe stage)
- The pipeline registers need to be wide enough to store all signals that need to propagate to subsequent stages.
- The pipeline control is similar to the single clock cycle control, except the signals are divided into groups and those needed by "downstream" pipe stages are sent via the pipeline registers.
- The control signals are generated when the instruction is at the ID pipe stage.
- The pipelined datapath is as follows:



* The register to be written is identified by the instruction currently at the WB stage and not the one at the ID stage (the data to write also given by the WB stage)

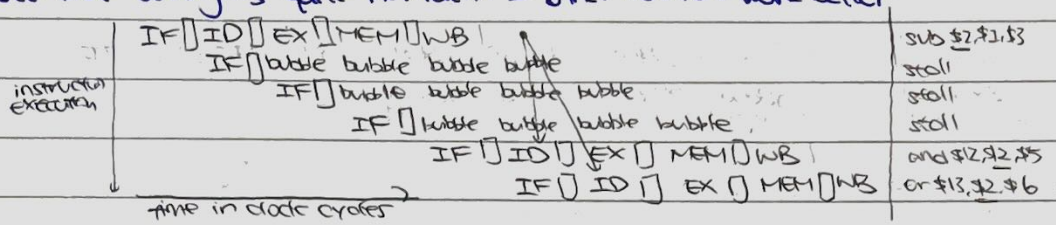
Data and branch hazards

- The instructions "flow" from left to right, w/ two exceptions:
  - ↳ write back to the register file
  - ↳ selection of the next PC value (branch)
- Pipelining is complicated by data and branch hazards:
  - ↳ Data hazards — value req. in a register is not valid because a previous instruction has not updated it yet (we read at stage 2 but write at stage 5)
  - ↳ Branch hazards — the address of the next instruction is not known (how to increment PC) is not known until the branch instruction reaches stage 4 of the pipeline.

↑ if the pipeline is not corrected, the wrong instructions will execute and write their results to memory or the register file.

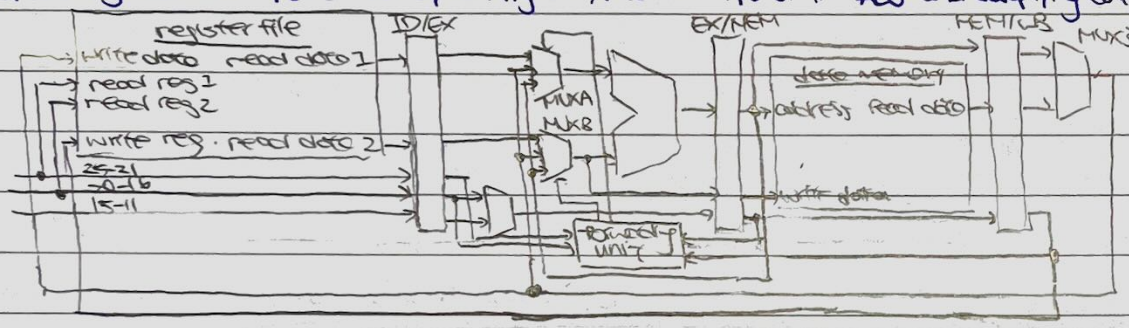
Data hazards — stalling the pipeline

- A solution to data hazards is stalling the pipeline — the ID stages of dependent instructions are not executed until the data req. is updated/available → introduce "bubbles" to pipeline
- stalling can be implemented by adding a hazard detection unit to the pipeline.
- If a hazard is detected, the unit stops the PC and IF/ID register from being written and insert zeros for the control values in the ID/EX register.
- Note that stalling is quite inefficient — other solutions work better

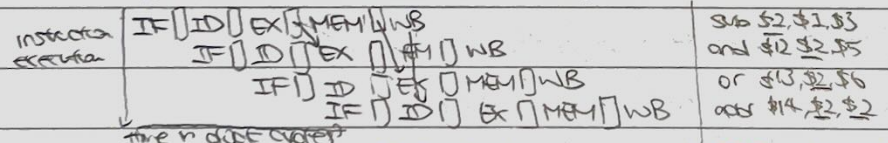


Data hazards — data forwarding

- A more complex way around data hazards is data forwarding — rather than wait for values to be available in the register file, the pipeline registers can be used.
- Forwarding can be implemented by adding MUX to the ips of the ALU and supplying control lines



- The forwarding unit compares the registers which have been just read (read reg 1, read reg 2) w/ only registers to be written by the two downstream instructions (write reg in EX/MEM and MEM/WB)
- If they are the same, MUXA, MUXB are set s.t. the value read from the register file is ignored and instead use the appropriate forwarded value (ALU op stored in EX/MEM or MEM/WB)



* we arrange st. in stage 2, write before read → we can do ADD w/o stalling.

- Data forwarding alone would not work when the instruction following a LW req. the result of the load.. we can get around this using one of the following:

trade-off btwn compiler and hardware complexity

↳ stall the pipeline — reintroduce the hazard detection unit (only 1 stall cycle req.)

↳ Req. software to follow the LW w/ an instruction independent of the load (delayed load)

In the worst case, the compiler places a NOP after the LW

* For the special case lw $8,0($9), sw $8,4($9) we could forward the LW result from the MEM/WB pipe register directly to the data memory's write data input but this req. further additions (hardware) to the datapath for just this case.

Branch hazards.

- There are several ways to deal w/ branch hazards:
  - ↳ Always stall on branches → penalty of several clock cycles
  - ↳ Assume branch not taken. Instructions after the branch start to execute. If the branch is in fact taken, the pipeline is flushed (partial results discarded) and execution resumes from the branch target address ← could have complex branch prediction based on whether the program previously branched there.
  - ↳ Delayed branches — instruction after a branch gets executed regardless → responsibility passed to programmer, compiler/hardware trade off. ← allows some useful work after branching w/o the risk of hazards.

Superpipelines and superscalar CPUs.

- A superpipeline is where a pipeline is split into shorter stages while increasing the speed at which it is clocked.
- This increases throughput, but hazards become more of a problem.
- Superscalar CPUs issue more than one instruction per clock cycle. Some hardware changes are:
  - ↳ Each pipeline register needs to be widened to handle multiple instructions.
  - ↳ We can still have one register file, but it needs to read and write more registers at the same time.
- There may be restrictions on instruction sequencing (eg only one load/store at a time)
- More sophisticated compilers are req. to reorder instructions s.t. as many instructions as possible can be executed in parallel w/o incurring data/branch hazards.
- The hardware may also have mechanisms to process later instructions while waiting for a stall to be resolved — this is dynamic pipeline scheduling.

Memory and IO systems

Caches

Memory hierarchy

- Memory is where programs and data are stored. Memory is arranged in a hierarchy:

	Type	Size	Access time	
	Registers	1KB	5ns	
SRAM →	Cache	8MB	10ns	↓ faster
DRAM →	Main memory (MM)	8GB	60ns	↓ cheaper
	Virtual memory (VM)	1TB	10ms	

- At each level of the hierarchy, the information stored is a subset of that lower down.
- Efficient operation relies on the principle of locality / locality of reference:
  - ↳ Temporal locality: items referenced tend to be referenced again soon (loops)
  - ↳ Spatial locality: neighbours of referenced items tend to be referenced soon (array)
- If requested data is present in an upper level (e.g. cache), this is a hit, o/w a miss.

Caches

- The cache is the level of the memory hierarchy btwn the CPU and the MM.
- Words in a cache are arranged in blocks to take advantage of spatial locality.
- A well-designed cache will achieve a low miss rate and a low miss penalty. Both of these quantities are influenced by the block size and the degree of associativity.
- Blocks can be arranged w varying degrees of associativity:
  - ↳ Direct-mapped:
  - ↳ Fully associative:
  - ↳ Set associative:
- A cache index points to a cache block (direct-mapped) or cache set (set-associative), and can be calculated using the MM address.

index = (block address) mod (no. of blocks in cache) note that computationally, $f(x) = x \bmod 2^n$ → take n LSB

where block address = $\left\lfloor \frac{\text{word address}}{\text{no. of words per block}} \right\rfloor = \left\lfloor \frac{\text{byte address}}{\text{no. of bytes per block}} \right\rfloor$ $g(x) = \left\lfloor \frac{x}{2^m} \right\rfloor$ → shift right by m bits

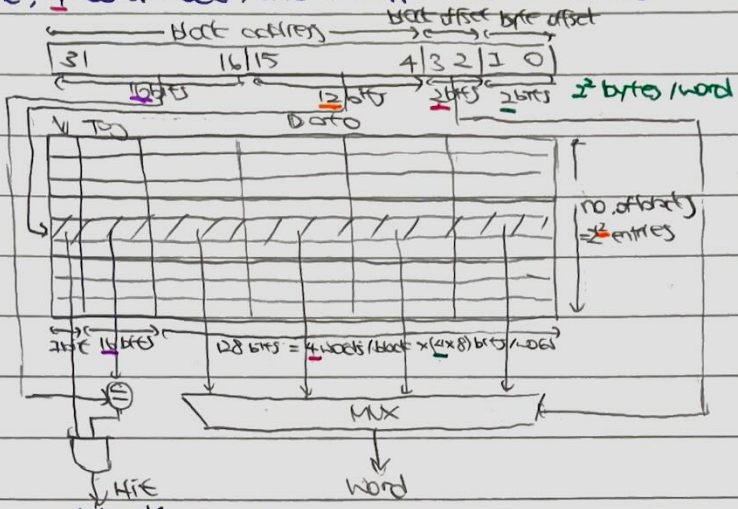
e.g.: 8-block cache, 4-word blocks. (block size = 4-word blocks)

cache index		block address	word addresses	byte addresses
0	←	0	0-3	0-15
1		1	4-7	16-31
2		8	32-35	128-143
3	← many-to-one mapping	9	36-39	144-159
4		6	24-27	96-111
5	←	7	28-31	112-127
6		12	48-51	192-207
7	←	13	52-55	208-223

- Increasing block size can reduce the miss rate (↑ spatial locality of reference), but increase the miss penalty (↑ no. of blocks to transfer)
  - ↳ only slightly as most of memory transfer time is a const. overhead.

Direct-mapped cache

- A direct-mapped cache allows a memory block to reside only in a single cache block.
  The cache index points to a cache block — no searching req.
- A 64 KiB cache, 4-word blocks, direct-mapped is as follows:

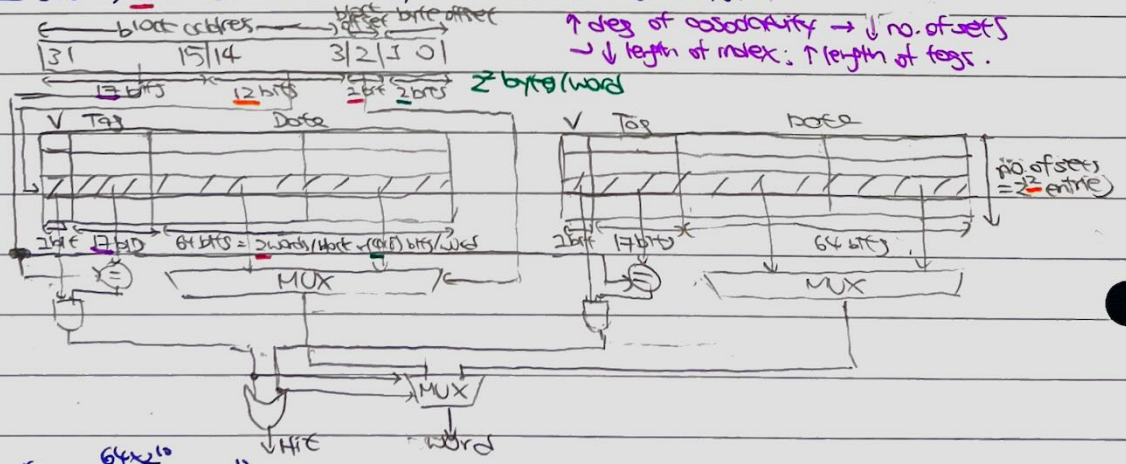


* no. of blocks = $\frac{64 \times 2^{10}}{4 \times 4} = 2^{12}$ → 12 bit cache index.

- To reference a word in the cache, the 12-bit index addresses the cache block, and if the 16-bit tag of the address matches the stored tag, the 2-bit block offset is used to get the req. word.
  (Tag req. because 2^{16} blocks can map to the same cache index).
- There is a valid bit associated w each cache block which is set to 0 (invalid) on start up.
- We have no choice which block to replace when we get a cache MISS (a MM block can only reside in one cache location). — inefficient if we replace the same block repeatedly.

Associative cache

- A fully-associative cache allows a memory block to reside anywhere in the cache.
  There is no cache index — all blocks in cache must be searched → usually too complex.
- A set-associative cache allows a memory block to reside in a no. of locations (≥ 2).
  The cache index points to a cache set — all blocks in set must be searched.
- A 64 KiB cache, 2-block words, 2-way set associative is as follows:



* no. of sets = $\frac{64 \times 2^{10}}{2 \times 2 \times 4} = 2^{12}$ → 12 bit cache index

cache misses

- If there is a cache miss, the req. block needs to be copied from MM into the cache and an existing block is replaced. The seq. of actions by the CPU control are as follows:
  - 1) compute PC+4 and store in the PC (reset PC to pt. to the instruction which missed)
  - 2) instruct MM to read the req. block and wait to complete (several clock cycles)
  - 3) write the data and tag into the cache and set the valid bit to 1.
  - 4) Restart the pipeline → the instruction will have a hit this time.
- All of the above is handled by the processor hardware.
* A cache miss causes a stall similar to a pipeline stall but simpler:
  - cache miss: we stall the entire machine while we wait for memory
  - pipeline stall: we continue executing some instructions while stalling others.

cache writes

- On writing, if the block is not already in the cache, the block must be read from MM, then the relevant word is updated in the cache.
- The MM must be updated on a write w/ one of the following approaches:
  - write through: Data is written to the cache and also to MM. To avoid stalling the processor while MM is being updated, we use write buffers to queue writes to MM.
    - how so? ✓: cheaper read miss (never req. a write to MM), easier to implement.
  - write back: Data is written only to the cache. The modified block is only written to MM when it is replaced in the cache.
    - how so? ✓: words written at cache speed, multiple writes to a block only req. 1 write to MM.

Block replacement strategies.

- We have a choice of which block to replace on a miss for associative caches
- There are two common block replacement strategies:
  - Random: candidate blocks are randomly selected
  - Least recently used (LRU): the block replaced is the one unused for the longest time.
- Random selection is easy to build into hardware, less so for LRU.
- As the size of cache increases, LRU becomes more expensive and only approx in practice.
- LRU becomes more effective w/ larger degree of associativity, but is slower and more difficult to implement.
- If the miss penalty is large, then some form of LRU is used since a small reduction in miss rate can make a big diff. to performance

Cache performance.

- Overall system performance can be factorised as follows:

  execution time = (CPU execution cycles + memory stall cycles) × clock cycle time

  where memory stall cycles = instructions in program × MISSES per instruction × MISS penalty.

- As CPUs get faster, the memory cycles have an increasing effect on overall performance.

  We can reduce the memory stall time by:

  ↳ Reducing the MISS rate w/ better cache strategies: use multi-level cache (small, fast primary cache, w/ low associativity + large secondary cache w/ higher associativity)

  ↳ Reducing the MISS penalty w/ faster memory: increase the BW to MM w/ a wider memory bus and DDR SDRAM technology.

Virtual memory

Magnetic disks.

- The most common form of non-volatile, long-term storage is the magnetic disk.
- Hard disks contain glass/ceramic platters (ϕ=1.8-3.5in) coated w thin film magnetic media.
- The platters rotate at speeds of 7200 rpm inside a sealed unit. Electromagnetic R/W heads float just above each surface. (each platter has 2 surfaces).
- Each surface is divided into ~280k tracks. The tracks are divided into 850-2600 sectors, which are the smallest units which can be read/written.
- The collection of tracks under the heads at a particular head position is a cylinder.
- The time to read/write a sector of data to/from the disk is given by

  read/write time = seek time + rotational latency + transfer time + controller overhead.

  ↳ seek time: time to position the arm over the right cylinder (avg=9ms)

  ↳ rotational latency: time for the right sector to rotate under the R/W head. (avg = $\frac{1/2 \cdot 60}{7200}$ = 4.17ms)

  ↳ transfer time: function of sector size, recording density and rotation speed (avg = 2.6ms)

  ↳ controller overhead: disk controller overhead (avg = 0.5ms)

- The time to read/write a sector seems to be dominated by the seek time, but we rarely incur the full seek penalty in practice due to locality of reference to the disk.

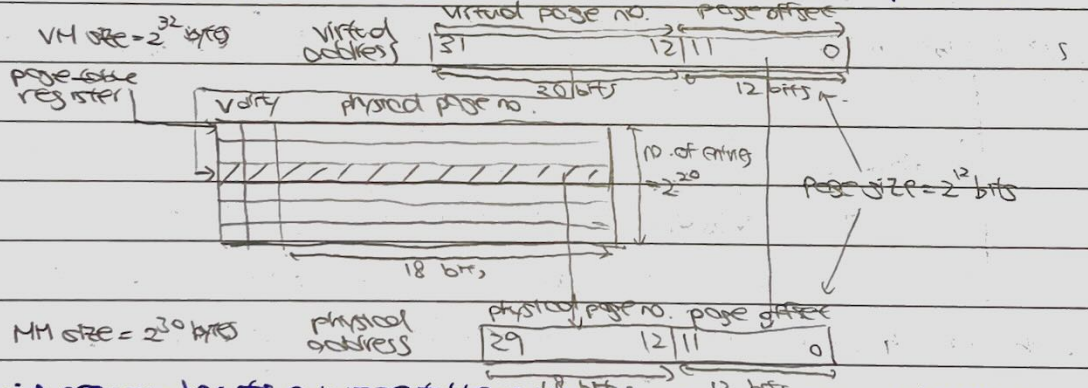
  (Files usually stored on contiguous cylinders w/ successive reference → track-to-track seek times are approx. 10% of average seek times)

Virtual memory (VM)

- VM allows the total memory available to a program to exceed the physical amount of MM.
- It also allows MM to be shared btwn multiple programs on the same machine: each program has its own address space and is protected from other programs.
- W/ VM, programs use virtual addresses — these are translated by hardware and the OS software into physical addresses, which are used to access MM.
- Both VM and MM are divided into chunks called pages. Since the no. of VM pages can exceed the no. of physical pages, some VM pages may be stored on disk.
- If the req. page is not in MM, then the translation will fail, generating a page fault. The page will have to be fetched from disk into MM, replacing another page in MM.
* MM ↔ cache. VM ↔ MM, page fault ↔ cache miss, page ↔ cache block.
- Most design choices of VM systems are motivated by the high miss penalty (disk access take hundreds of thousands of clock cycles):
  - ↳ pages tend to be quite large
  - ↳ Flexible replacement of pages is essential (fully-associative page placement)
  - ↳ Misses handled by OS → sophisticated page replacement strategies possible (LRU).
  - ↳ Write through not possible as we want to min. the no. of disk writes.

Page tables

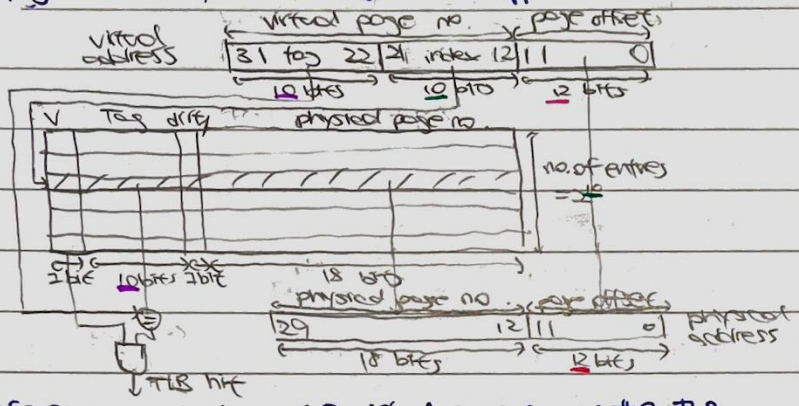
- VM systems use fully-associative page placement. The location of pages is stored in a page table, indexed by the virtual page no. and contains the physical page no.



- Each running program has its own page table in MM. The memory system maintains the page-table register to pt to the start of the page table.
- For each page table entry, there is a valid bit to determine whether the page is in MM. If a page is req. but the valid bit is zero, a page fault occurs. (OS implements LRU page replacement)
- For writes, write-back is used. To save writes when a page hasn't changed, the page includes a dirty bit to show that the page's contents differ from the data on the disk. (The dirty bit is set the first time the page is modified in MM → needs copying back when replaced in MM)

Translation lookaside buffers (TLB)

- page tables are large and need to be stored in MM → slows down memory access dramatically. (even if the word is in cache, we still need to access the page table in MM to get the physical address).
- To get around this, we exploit locality of reference to the page table and build a special, fast cache to store recently used translations — the TLB. (dedicated cache, not GP cache) ↳ very effective as we can have thousands of ref on the same page.
- The TLB caches only those portions of the page table which points to MM addresses
- Each tag entry in the TLB holds all (fully-associative) or part (set-associative, direct-mapped) of the virtual page no. and each data entry contains a physical page no., TLB valid bit and dirty bit.
- A 1024-entry TLB, 1 page table entry per block, direct mapped is as follows: (block size)



- On every memory reference, we look up the virtual page no. in the TLB.
  - ↳ Hit: we can rapidly generate the physical address from the TLB and access the cache.
  - ↳ Miss: if page exists in MM, the TLB entry is updated from the page table, retry translation otherwise, a page fault is generated.
- The TLB is often fully-associative w/ random replacement. write back is used to ensure that the dirty bits in the page table are kept up to date.
- * TLB works like standard cache, except it stores page table entries and not arbitrary memory words.

Processes

- The state of a program (process) is defined by the page table, PC and other registers.
- The OS can allocate the CPU to another process by saving the current state and retrieving the state of the new process, which then starts to execute.
- When a process switch occurs, we need to flush the TLB (set all TLB valid bits to zero) or extend the virtual address w/ a process identifier to distinguish entries from diff. processes.
- The VM system automatically enables process protection — each process can only alter the physical pages in its own page table. The page tables are kept organised by the OS s.t. independent virtual pages map to disjoint physical pages.
- Only OS processes (not user processes), called kernel / supervisor processes can modify the page tables.

Interfacing processors and peripherals

Input/output (I/O).

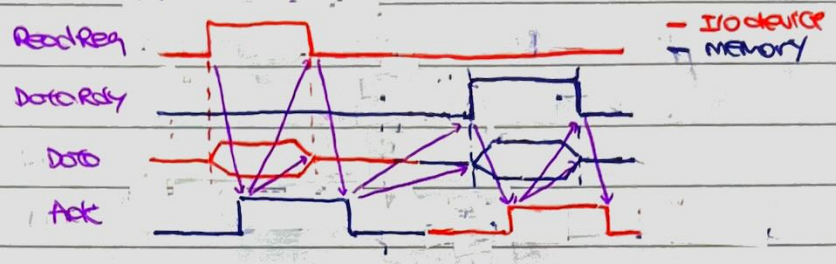
- I/O performance can be measured by:
  - ↳ BW: how much data can flow through the system per second
  - ↳ Latency: how long does it take to initialise an I/O process
- If the I/O req. are large, response time will depend heavily on BW; for small accesses, the I/O system w/ the lowest latency will deliver the best response time.
- I/O devices vary widely in their behaviour (I/p/o/p/storage) and data rates (0.01 KB/s – 8GB/s)

Buses.

- I/O devices can be connected to the processor using a bus (shared communications link which uses one set of wires to connect multiple subsystems)
- A bus usually has a set of data lines (data/address transfers) and a set of control lines (req./ack signals).
- A disadvantage of a bus is that it can create a potential bottleneck. It is difficult to make buses both fast and long.
- There are two main classes of bus:
  - ↳ processor-memory: short, high performance, matched memory system (e.g. DDR4)
  - ↳ I/O: long, wide range of data rates, used to connect diverse devices (e.g: USB 3.0)
- There are two schemes for bus communication:
  - ↳ synchronous: control lines include a clock signal, all transfers are rel. to the clock.
  - ↳ Asynchronous: no clock, every transfer is acknowledged by recipient before more data is sent.
- Synchronous buses must run at the speed of the slowest device on the bus (due to shared clock). clock skew (the diff. in seeing the same clock edge at diff. positions on the bus) limits bus length.
- Synchronous buses can run very fast and are used for processor-memory buses – devices communicating are close together and run at high clock speeds. Further speed up possible w/ separate data/address lines.
- Asynchronous buses can cope w/ a wide variety of devices and data rates, and can be lengthened w/o worrying about clock skew.
- Asynchronous buses use a handshaking protocol to coordinate data transfers.

Handshaking

- A simple handshaking protocol req. three control lines:
  - ↳ ReadReq: indicates a read req, and we put the address on the data lines at the same time
  - ↳ DataRdy: indicates that data is available on the data lines
  - ↳ Ack: acknowledges the ReadReq or DataRdy signal of the other party.
- An example of a bus transaction is as follows:



Memory mapped I/O

- I/O events are typically presented to the processor using memory-mapped I/O. — each device appears as a set of data and control registers at a particular address.
- The control register contains various status bits which can be examined.
- Intuitively, a processor can continually check the status registers to see if an I/O event needs servicing (polling). This can be wasteful of CPU time and only used for low BW devices.
- A more efficient alternative is interrupt-driven I/O — an I/O device generates an interrupt (via control lines on the CPU) when it req. attention.
- When the CPU sees an interrupt, it stops executing the current process, finds the cause and services it. (determine which device)

Direct memory access (DMA).

- Interrupt-driven I/O can be a huge load on CPU for high BW devices, which can generate interrupts at a high rate → we can use DMA — transfer data to/from memory w/o involving the CPU.
- A DMA controller becomes the bus master (initiate and control all bus transactions) while data is being transferred.
- The CPU is still interrupted, but only at the start and end of each DMA transfer, not on every bus transaction. The steps in a DMA transfer are as follows:
  - ↳ 1) The processor sets up the DMA by providing the controller the identity of the device, the operation to perform, the source/destination memory address and no. of bytes to transfer.
  - ↳ 2) The DMA controller starts the operation and acquires the bus. The transfer of the data block is performed. After each transaction, the DMA controller updates the memory read/write addresses.
  - ↳ 3) Once the DMA transfer is complete, the DMA controller interrupts the CPU, which passes back bus master duties to the CPU. The CPU then checks the operation completed successfully.

DMA, virtual memory and cache

- A DMA process req. the start address and no. of bytes to transfer.
- If the DMA is given a physical address, then all DMA transfers must be constrained to stay within one page.
- If the DMA is given a virtual address, then the necessary page mappings / translations must be provided for the transfer.
- For both approaches, the OS must cooperate by not moving pages around while a DMA transfer involving that page is in progress.
- DMA poses two major problems to cache systems regarding cache coherency:
  - ↳ If the DMA places data directly from disk into memory, we could have stale data in the cache, which the processor receives the next time it reads from cache.
  - ↳ If we have write-back cache, we could have stale data in the memory, which may be transferred to disk by the DMA unit.
- We can get around this using the following:
  - ↳ Route all DMA activity through the cache — generally wasteful of cache space
  - ↳ Have the OS invalidate the cache for an I/O write, or force write-backs for an I/O read (cache flushing)
  - ↳ Selectively flush / invalidate cache entries for DMA (best of'n but most complex).

exactly what?

From buses to networks

- There is a general trend towards replacing parallel I/O bus w/ serial point-to-point networks:
- Buses cannot keep up w/ the BW of today's I/O devices. The clock rate is limited by noise, stray capacitance, cross talk and clock skew.
- To make a fast parallel bus, we would need to make it short and limit the no. of devices connected to it — this conflicts the basic req. that an I/O bus is exp / supports many devices.
- This is solved using serial point-to-point networks. The networks are serial → mitigate effects of diff. timing skews on each bit; only two devices on each link → less load, noise and stray capacitance so fast
- The new I/O networks also circumvent the need for bus mastering protocols. Data transfer is synchronous to an embedded clock (e.g. 8b/10b encoding).
- Serial connections also req. fewer wires → less clutter and hence better cooling possible.
- Examples of the transition from buses to networks include:
  - ↳ Disk devices: Parallel ATA → Serial ATA
  - ↳ Generic I/O devices, graphics cards: PCI → PCI express.

Parallel processing.

Flynn's classification

- The following forms of parallelism are possible:
  - ↳ SISD: single instruction stream, single data stream — uniprocessor machine
  - ↳ SIMD: single instruction stream, multiple data streams — vector, array operations
  - ↳ MISD: multiple instruction streams, single data stream
  - ↳ MIMD: multiple instruction streams, multiple data streams — most general parallel processor.

* A SISD has some parallel operations: the 32-bit ALU processes 32 bits in parallel, pipelining overlaps several instructions in parallel (superscalar processors have several pipelines in parallel)

SIMD computers

- SIMDs are at their best when dealing w/ arrays and vectors. The application has to be data parallel for SIMDs to be effective.
- The key features of SIMD computers are:
  - ↳ A single SISD host unit. PC steps through the instructions (only one copy of the program)
  - ↳ A typical program includes both SISD and SIMD instructions. SIMD instructions are broadcasted to all execution units over a network.
  - ↳ Each execution unit has its own registers and memory. They are synchronised — they respond to the same instruction from the host at the same time, but operate on different data
- SIMD was first used in vector supercomputers (e.g. Connection Machine 2, Illiac IV). Small scale SIMD units were then built into general purpose CPU and used by instruction set extensions. Now, dedicated graphics processors w/ embedded SIMD take over this task from the CPU (graphics, ML)

MIMD computers

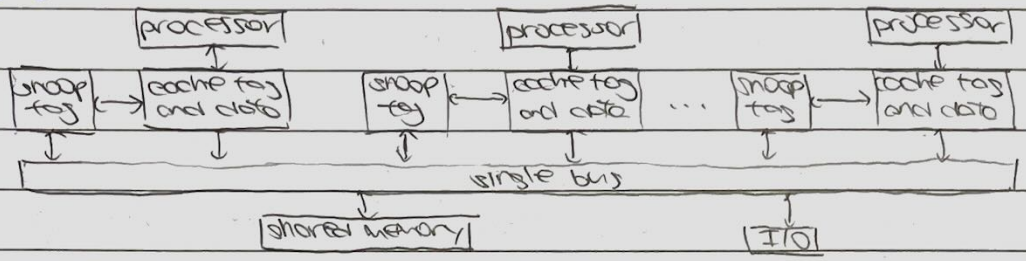
- MIMD are essentially n SISD machines connected on a single bus or via a network. MIMD machines offer scalable hardware and resilience to partial hardware failure.
- MIMDs can be used to run a separate process on each processor (easy) or a parallel processing program — a single program running on multiple processors simultaneously (hard)
- The processors of a parallel processing program can communicate in two ways:
  - ↳ Single address space: the processors share memory, communicate via loads and stores, synchronisation req. btwn processors
  - ↳ Message passing: each processor has its own private memory, communicate by passing messages, synchronisation is implicit.

* For single address space, if a memory access takes the same time regardless of the processor requesting, we have uniform memory access (UMA) / symmetric multiprocessor (SMP), o/w, we have non-UMA (NUMA)

- Effective parallel processing programs are hard to develop because:
  - ↳ communications overhead / cache issues limit speedup.
  - ↳ The programmer needs to know a lot about the hardware to exploit it.
  - ↳ It is not much use parallelising small parts of a program.

MIMDs connected by a single bus.

- A typical single-bus UMA contains 2 to 32 processors. The bus is a bottleneck, so caches are used to lower the traffic → we req. mechanisms to enforce cache coherency.
- The most common technique is bus-snooping — all cache controllers monitor the bus and reacts to read/writes in other caches.



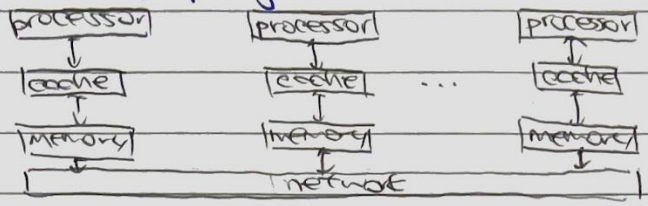
- There are two ways of dealing w/ writes:
  - ↳ write-invalidate: The writing processor broadcasts an invalidate signal to all other caches. If they have a copy of the same block, they invalidate it immediately
  - ↳ write-update: The writing processor broadcasts the new data over the bus. If other caches have a copy, they update it immediately

→ write-invalidate only uses the bus on the first write to invalidate all other copies; write-update uses the bus for all writes to update copies of the shared data (like write-through)

- Practical UMA systems tend to use write-back caches and write-invalidate cache coherency protocols to reduce bus traffic → more processors can be connected to the bus.
- On a read MISS, the cache must notify all other caches (rather than going straight to MM), as another cache may have a more up-to-date copy of the data (for write-back cache)
- Coherency protocols (e.g. MSI) do not interfere much w/ the processor. The address tags are duplicated in the cache for exclusive use by the snoop controller → processor has full access to normal portions of the cache.
- Single bus designs are limited by conflicting bus req. of high BW, low latency and large length.
  → not possible to connect more than few tens of processors on a single bus.
- Single memory systems are also limited by BW.

MIMDs connected by a network

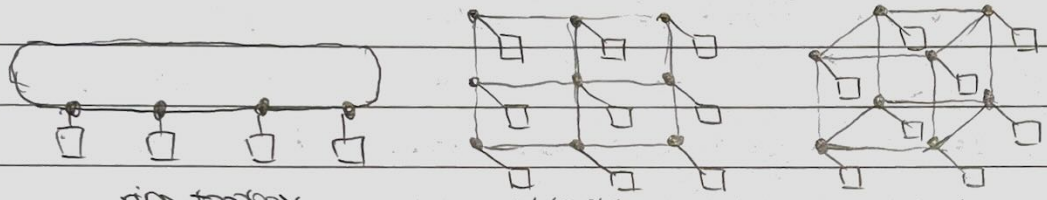
- To connect more processors, we use a network, and provide each processor w/ local memory (to reduce network traffic).
- Such systems have distributed memory and multiple private address spaces w/ communication by message passing.



- An alternative is to use a global address space w/ cache-concurrency enforced w/ directories.

Network topologies

- The performance of network-connected MIMDs is highly dependent on the design of the network.
- At the other end of the scale from the bus is the fully-connected network — dedicated link btwn every pair of nodes → best performance but expensive and can be infeasible.
- Btwn the two extremes is a wide range of possible topologies:



- A network is capable of many simultaneous transfers. We can measure its performance using:
  - ↳ Total network BW: The network's best possible performance, when all nodes talk to their neighbours. Total network BW = no. of links × BW per link.
  - ↳ Bisection BW: Closer to the worst case. Divide the network into two, then sum the BW of the links crossing the division.
- For a network w/ n processors, and each link has a BW of b,
  - ↳ Bus: 1 link → TNBW = b, BBW = b
  - ↳ Ring: n links → TNBW = nb, BBW = $2b$
  - ↳ Fully-connected: C_2^n links → TNBW = $\frac{n(n-1)}{2}b$, BBW = $\left(\frac{n}{2}\right)^2 b$
- → The selection of a suitable network is a trade off btwn performance and cost.

Large scale NUMAs and clusters

- The Silicon Graphics Origin 3000 series has 512 processors and 1TB memory. It is a NUMA machine w/ hypercube interconnection network → very expensive as sales volume very low.
- An alternative approach to large scale parallel processing is to use clusters of off-the-shelf PCs.
  there are two classes of clusters: (req. fast intercommunication btwn threads (eg infiniband))
  - ↳ High performance computing (HPC) clusters ← thread level parallelism
  - ↳ Warehouse scale computer (WSC) clusters — request level parallelism
- A comparison btwn large NUMAs and clusters is as follows:

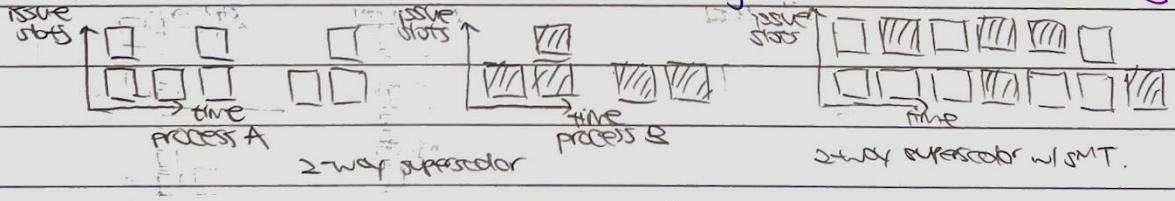
Network-connected NUMA MIMD	cluster
- connected via memory bus	- connected via I/O bus
- fast hardware load/store comm.	- slow software send/receive comm.
- one OS and machine to administer	- many OS and machines to administer
- one program can use all memory	- one program can only use one PC memory
- cannot replace processor w/o shutting down	- can replace PC w/o shutting down cluster
- Expensive	- cheaper

- The key to the reliability of Google's WSC is redundancy:
  - ↳ multiple sites w/ indep. internet services
  - ↳ backup electricity generators
  - ↳ more network BW and CPU than req.
  - ↳ private links btwn the sites
  - ↳ redundant ethernet switches
  - ↳ multiple copies of search index and page cache.

(even though they use ordinary PCs — 1-2 SATA drives, a few GB SDRAM, Intel CPU, Linux OS).

Simultaneous multithreading (SMT)

- Superscalar processors (SISD) have more capacity than a single process can effectively use (many instruction issue slots unfilled w/ dynamic pipeline scheduling, cache misses, data hazards)
- By allowing two (independent) processes to run concurrently, w/ instructions fetched in the same cycle, the superscalar hardware spends less time idling — this is SMT. (not req. process switching. Intel calls this hyperthreading.)



- For two (independent) processes to share the main functional units of a superscalar CPU, we req. a separate register file, PC, TLB to accommodate the independent state for each process

* SMT is diff. from multi-core CPUs which have multiple processors on the same chip. They are UMA MIMDs. However each processor can be SMT.