1 Basics of Logic

1.1 Gates, truth tables, and logic equations

- Signals are either true/1/high voltage, or false/0/low voltage
- **Combinational** block: lack memory and depends only on input
- **Sequential** logic: includes block with *state*, *i.e* memory
- *Truth table* define outputs for each possible set of input (2^n)

A	B	C	out	
0	0	0	0	$\bar{A}\,\bar{B}\bar{C}$
0	0	1	1	$\bar{A}\,\bar{B}C$
0	1	0	1	$\bar{A}B\bar{C}$
0	1	1	0	$\bar{A}BC$
1	0	0	1	$A\bar{B}\bar{C}$
1	0	1	1	$A\bar{B}C$
1	1	0	0	$AB\bar{C}$
1	1	1	0	ABC

Sum of products: Out = $\bar{A}\bar{B}C + \bar{A}B\bar{C} + A\overline{BC} + A\bar{B}C$

1.1.1 Boolean algebra

Express the logic function with logic equation. All the variables have the values 0 or 1 and there are 3 operators:

- OR ($A + B$) or logical sum
- AND ($A \cdot B$) or logical product
- NOT ($\bar{A}$) or inversion/negation

1.1.2 Gate

Logic blocks are built from *gates* that implement basic logic functions. AND and OR gates are commutative and associative → can have multiple inputs.

AND gate, OR gate, and an inverter. Rather than explicitly drawing inverters, can add bubbles to the inputs or outputs of a gate.

- Can build any logical function using AND, OR, and inversion
- Any logic function can be represented as a sum of products (canonical forms). 💡 Sum all true output entry of truth table, which is essentially a product term.
- NOR, NAND are universal: can build any function using one type

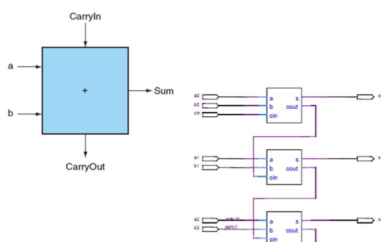
1.2 Combinational logic

- **Encoder/decoder**: n-bit input ↔ 2^n output (one-hot)
- **Multiplexors**: choose one of input values depending on a *selector* (or *control*) value
  - Can construct with decoder, n AND gates, and single OR to $(In_1 \cdot Out_1) + \cdots + (In_n \cdot Out_n)$

1.2.1 A full adder

Can build multiple bit adder by connecting 1-bit adder.

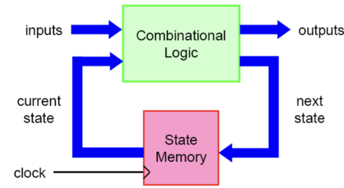
A	B	CIN	SUM	COUT
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



(a) 1-bit adder

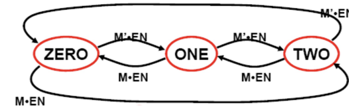
(b) Ripple-carry adder

1.3 State machine



State machine.

Computer is a finite state machine (FSM).



Modulo-3 counter

- Counter enabled when `EN`=1
- `UP` or `DOWN` mode: decrease counter when `M`=1
- State is 2 bits (00, 01, 10)

1.4 Elements

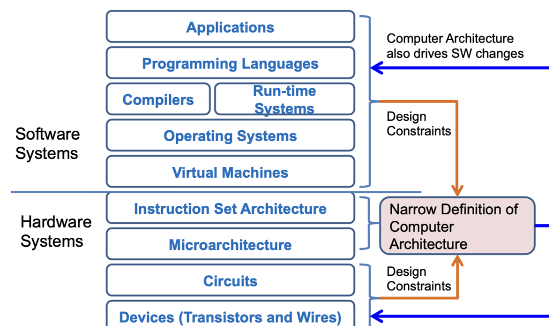
- **Clock** is free-running signal with a fixed cycle time.
- Latch: state changes when the enable is asserted
- Flip-Flops: state change only on a clock edge (rising or falling).

Register is collection of flip-flops connected in parallel. Primary state-holding elements in a processor.

2 Introduction

2.1 6 great ideas in computer architecture

- **Abstraction**: Instruction set *arch* (ISA) is HW/SW interface
- **Moore's law**: transistors on single chip 2× about every 2 years
  - Transistors: electrical switch/building block of integrated circuits
  - Faster/smaller transistors → clock frequency improvement
  - Instruction-level parallelism (ILP)
- ⚠ Almost still valid but power and heat limits clock frequency
- **Memory hierarchy** (Principle of **locality**)
- **Parallelism**: data-, memory-, and instruction-level
  - HW systems are inherently parallel
- **Performance measurement**: determine the design based on performance, cost, and design complexity
- **Dependability**/reliability via **redundancy**



Computer system abstraction

2.2 Computer architecture

Instruction set architecture (ISA)

- Interface between HW and SW systems
- Hard to change due to compatibility issues
- Various ISAs: MIPS, x86, Power, ARM, RISC-V, etc
- ISA evolves: Intel and AMD have been extending x86 (32 and 64bit support, VM support)

Microarchitecture is implementation of ISA and affects performance.

2.3 Performance

What factors in the architecture contribute to overall system performance and the relative importance (and cost) of these factors?

- **Throughput** (bandwith): the time between the start and the completion of the task → interest of individual users
- **Response time** (execution time): total amount of work done in a given time → interest of managers

2.3.1 Defining performance

Performance = 1 / Execution time

Operation of digital hardware is governed by a constant-rate clock.

- Clock period: duration of a clock cycle (CC)
- Clock frequency/rate (CR): cycles per second

CC = 1/CR

- CPU execution time (CPU time): time the CPU spends working on a task = # CPU clock cycles / CR
- Clock cycles per instruction (CPI): the average number of clock cycles each instruction takes to execute

CC = # instructions * CPI

$$\therefore \text{CPU time} = \frac{\text{Instruction count} \times \text{CPI}}{\text{CR}}$$
$$\text{or} = \text{Instruction count} \times \text{CPI} \times \text{CC}$$

→ separate the three key factors that affect performance.

- Can measure CPU time by running the program
- CR is usually given
- Can measure overall instruction count by using profilers or simulators without knowing all of the implementation details
- CPI varies by instruction type and ISA implementation for which we must know the implementation details

2.3.2 Amdahl's law

Improving an aspect of a computer and expecting a proportional improvement in overall performance.

$$T_{\text{improved}} = \frac{T_{\text{affected}}}{\text{improvement factor}} + T_{\text{unaffected}}$$

Consider a program which A takes 80s and B takes 20s. We cannot make a program 5× faster by improving only A.

Op	Freq	CPI_i	Freq x CPI_i			
ALU	50%	1	.5	.5	.5	.25
Load	20%	5	1.0	.4	1.0	1.0
Store	10%	3	.3	.3	.3	.3
Branch	20%	2	.4	.4	.2	.4
			Σ = 2.2	1.6	2.0	1.95

- **How much faster would the machine be if a better data cache reduced the average load time to 2 cycles?**
  CPU time new = 1.6 x IC x CC so 2.2/1.6 means 37.5% faster
- **How does this compare with using branch prediction to shave a cycle off the branch time?**
  CPU time new = 2.0 x IC x CC so 2.2/2.0 means 10% faster
- **What if two ALU instructions could be executed at once?**
  CPU time new = 1.95 x IC x CC so 2.2/1.95 means 12.8% faster

2.4 Multiprocessors

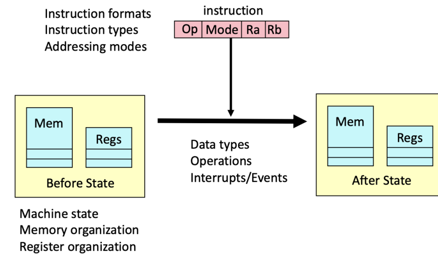
Explicit parallel programming. ↔ instruction level parallelism (ILP).

- Hardware executes multiple instructions at once
- Hidden from the programmer
- Load balancing, Optimizing communication and synchronization

3 Instruction set architecture

Contract between programmer and the hardware. Defines visible state of the system and how they changes in response to instructions. ISA specifies:

- All architecturally visible states: registers, condition codes, etc
- Instruction formats and behaviors
- Memory model (paging, segmented memory, etc)
- IO (interrupts)



3.1 MIPS-32 ISA

- Instructions: computational, load/store, jump, branch, FP, memory
- 32 registers (R0-R31), PC, HI, LO
- Regularized: 3 instruction formats (R, I, J)

3.2 Operands

3.2.1 Number of explicit operands

- **3**: 1 results + 2 inputs *e.g.* R1 = R2 + R3 → `ADD` R1, R2, R3
- **2**: 1 result/input + 1 input *e.g.* R1 = R1 + R2 → `ADD` R1, R2
- **1**: implicit accumulator + 1 input *e.g.* acc = ac + R1 → `ADD` R1
- **0 (stack ISA)**: Add top 2 elements of stack → `ADD`

Number of operands affects instruction length.

3.2.2 Register operands

MIPS register file holds 32 registers.

- 5-bit address (or number) ∵ $2^5 = 32$, 32-bit write data
- 2 read ports, 1 write port

Registers are

- Faster than main memory (*locality*)
- Easier for compiler to use (↔ stack)
- Read/write port increase impacts speed quadratically → number of operands is limited

3.2.3 Memory operands

Main memory used for composite data *e.g.* array.

- Byte-addressed: each address identifies an 8-bit byte
- Words are aligned in memory: Address must be a multiple of 4
- MIPS is **big endian**: most significant *byte* at least address of a *word*

Operating on memory data requires load/store → more instructions

C code:

```
A[12] = h + A[8];
```

- h in $s2, base address of A in $s3

Compiled MIPS code:

- Index 8 requires offset of 32

```
lw  $t0, 32($s3)    # load word
add $t0, $s2, $t0
sw  $t0, 48($s3)    # store word
```

3.2.4 Immediate operands

```
addi $s3, $s3, 4
```

- No subtract immediate instruction. Use a negative constant.
- Constant can have less bit → may need sign extension before addition

3.2.5 The constant zero

Register 0 (`$zero`) is the constant 0 and cannot be overwritten. Useful for common operations *e.g.* move between registers `add $t2, $s1, $zero`. ∴ Move is a **pseudo-instruction** achieved by adding zero. (abstraction for human).

3.3 Representing instructions

Instructions are encoded as 32-bit instruction words, called machine code. Assembler names corresponds to *regs* as below.

- Temporary: `$t0-$t7` are registers 8–15, `$t8-$t9` are 24-25
- Saved variables: `$s0-$s7` are 16–23

3.3.1 R-format instructions

op	rs,rt+	rd+	shamt	funct
6 bits	5bits each	5 bits	5 bits	6 bits
opcode	1st/2nd source	destination	shift amount	function code

Function code extends operation code (`opcode`). +as an register number.

R-format example: `add $t0, $s1, $2`

op	rs	rt	rd	shamt	funct
special	$s1	$s2	$t0	0	add
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

3.3.2 I-format instructions

Immediate arithmetic and load/store instructions.

op	rs	rt	constant or address
6 bits	5bits	5 bits	16 bits

- `rt`: destination **or** source register number
- if constant: $-2^{15} \sim +2^{15}-1$
- if address: offset added to base address in `rs`

3.3.3 J-format instructions

op	address
6 bits	26 bits

3.4 Operations

3.4.1 Logical operations

MIPS	C	Operation
`sll, slr`	«, »	Shift left, right
`and/andi, or/ori, nor`	&,|, ~	Bitwise AND, OR, NOR

These shifts are logical: fill with 0 bits. NOT is achieved via `nor` *e.g.* `nor $t0, $t1, $zero`.

3.4.2 Conditional operations

Branch to a labeled instruction `L1` if a condition is true. Otherwise, continue sequentially.

- `beq rs, rt, L1`: if `rs == rt`
- `bne rs, rt, L1`: if `rs != rt`
- `j L1`: unconditional jump

Set result to 1 if condition is true, otherwise 0. Use in combination with `beq, bne`.

- `slt rd, rs, rt`: if `rs<rt` set `rd`
- `slti rt, rs, C`: if `rs<C` set `rt`
- Unsigned operations `sltu, sltui`

```
$s0 = 1111 1111 1111 1111 1111 1111 1111 1111
$s1 = 0000 0000 0000 0000 0000 0000 0000 0001
slt  $t0, $s0, $s1  # signed
  • -1 < +1 ⇒ $t0 = 1
sltu $t0, $s0, $s1  # unsigned
  • +4,294,967,295 > +1 ⇒ $t0 = 0
```

❓ Why not `blt, bge`? Hardware for $<, \geq$ slower than $=, \neq$.

- C code:

```
if (i==j) f = g+h;
else f = g-h;
```

  - f, g, … in $s0, $s1, …
- Compiled MIPS code:

```
      bne $s3, $s4, Else
      add $s0, $s1, $s2
      j   Exit
Else: sub $s0, $s1, $s2
Exit: …
```

(a) `if-else` compiled

- C code:

```
while (save[i] == k) i += 1;
```

  - i in $s3, k in $s5, address of save in $s6
- Compiled MIPS code:

```
Loop: sll  $t1, $s3, 2
      add  $t1, $t1, $s6
      lw   $t0, 0($t1)
      bne  $t0, $s5, Exit
      addi $s3, $s3, 1
      j    Loop
Exit: …
```

(b) `while` compiled

3.4.3 Basic block

A basic block is a sequence of instructions with

- No embedded branches (except at end)
- No branch targets (except at beginning)

A compiler identifies basic blocks for optimization.

3.5 MIPS (RISC) design principles

1. Simplicity favors regularity: fixed size *ins*, few *ins* formats
2. Smaller is faster: limited *ins*/*reg*/addressing modes
3. Make common case fast: arithmetic operands from *reg*, immediate *ins*
4. Good design demands good compromises: 3 *ins* formats

3.6 Assembling

- **Branching** is PC-relative. Once pseudo-instructions are replaced by real ones, we know how far it should jump
- **Forward reference problem**: Labels may exist forward in the program → take 2 passes, first to remember position of labels, second to generate code

Jumps require absolute address. References to static data can't be known while assembling a single file.

3.6.1 Symbol and relocation tables

- **Symbol table**: list of items that can be used by the code in this file and in other files
  - Labels: function calling
  - Data: global variables in the `.data` section
- **Relocation table**: list of items that this file needs from other object files or libraries
  - Any label jumped to: `j` or `jal`
  - Any piece of data in static section

3.7 Procedure calls

3.7.1 Stack basics

- Part of memory to store local variables in function
- Grows down
- Each procedure call creates a stack frame
- On function exit, remove the stack frame by moving SP

3.7.2 Execution of a procedure

1. Main routine (caller) places parameters in a place where procedure (callee) can access
   - `$a0-$a3`: four **argument** registers
2. Caller transfers control to the callee (SP)
3. Callee acquires the storage resources needed
4. Callee performs the desired task
5. Callee places the result value in a place where caller can access
   - `$v0-$v1`: two **value** registers for result
6. Callee returns control to the caller
   - `$ra`: return address register to return to point of origin = `PC+4`

Name	Register Number	Usage	Preserve on call?
$zero	0	constant 0 (hardware)	n.a.
$at	1	reserved for assembler	n.a.
$v0 - $v1	2-3	returned values	no
$a0 - $a3	4-7	arguments	yes
$t0 - $t7	8-15	temporaries	no
$s0 - $s7	16-23	saved values	yes
$t8 - $t9	24-25	temporaries	no
$gp	28	global pointer	yes
$sp	29	stack pointer	yes
$fp	30	frame pointer	yes
$ra	31	return addr (hardware)	yes

MIPS register convention

3.7.3 Procedure instructions

Procedure call `jal ProcedureAddr`

- Save `PC+4` in register `$ra` procedure return
- Jumps to target address
- J format: `op=0x03`

Procedure return `jr $ra`

- Copies `$ra` to PC
- Can also be used for computed jumps (switch statement)
- R format: `op=0x0, rs=0x1f, funct=0x08`

3.7.4 Spilling registers

What if the callee needs to use more registers than allocated to argument and return values? → use stack

- Push: `$sp = $sp-4`, data on new `$sp`
- Pop: Get data at `$sp`, `$sp = $sp+4`

3.7.5 Leaf procedure example

```
int leaf_example (int g, h, i, j) { // in $a0,...$a4
    int f; // in $s0, save on stack
    f = (g+h)-(i+j);
    return f; } // result in $v0
```

```
leaf_example:
  addi $sp, $sp, -4        Save $s0 on stack
  sw   $s0, 0($sp)
  add  $t0, $a0, $a1       Procedure body
  add  $t1, $a2, $a3
  sub  $s0, $t0, $t1
  add  $v0, $s0, $zero     Result
  lw   $s0, 0($sp)         Restore $s0
  addi $sp, $sp, 4
  jr   $ra                 Return
```

3.7.6 Non-leaf procedures

- Procedures that call other procedures
- For nested call, caller needs to save on the stack
  - its return address
  - any arguments and temporaries needed after the call
- Restore from the stack after the call

```
int fact (int n) {
    if (n<1) return 1;
    else return n * fact(n-1); }
```

```
fact:
     addi $sp, $sp, -8     # adjust stack for 2 items
     sw   $ra, 4($sp)      # save return address
     sw   $a0, 0($sp)      # save argument
     slti $t0, $a0, 1      # test for n < 1
     beq  $t0, $zero, L1
     addi $v0, $zero, 1    # if so, result is 1
     addi $sp, $sp, 8      #   pop 2 items from stack
     jr   $ra              #   and return
L1:  addi $a0, $a0, -1     # else decrement n
     jal  fact             # recursive call
     lw   $a0, 0($sp)      # restore original n
     lw   $ra, 4($sp)      #   and return address
     addi $sp, $sp, 8      # pop 2 items from stack
     mul  $v0, $a0, $v0    # multiply to get result
     jr   $ra              # and return
```

3.8 Memory addressing modes

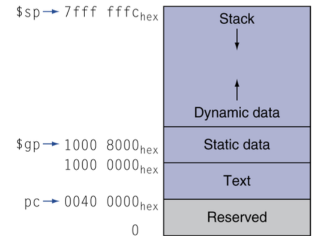
How to specify memory location = calculate effective address

- **Immediate**: `R4` ← `Imm` (read `Imm` and save it to register)
- **Register indirect**: `R4` ← `Mem[R1]`
- **Displacement**: `R4` ← `Mem[R1+Disp]`
- **Indexed**: `R4` ← `Mem[R1+R2]`
- **Memory indirect**: `R4` ← `Mem[Mem[R1]]`
- **Autoincrement**: `R4` ← `Mem[R1]`, `R1` ← `R1+d`
- **Scaled**: `R4` ← `Mem[R1+R2*scale]`

Effect on instruction count → size of binary (*instruction footprint*). MIPS only support displacement mode. x86 support displacement, indexed, and scaled mode.

3.9 Memory layout

- Text: program code
- Static data: global variables
- Dynamic data: heap
- Stack: automatic storage



3.10 Misc.

Character data. Byte-encoded character sets. ASCII (8bit) contains 128 characters. → **Byte/halfword operations**

- **`lb rt, offset(rs)`** load `offset(rs)`, take 1byte and sign-extend
- **`lbu`** does zero-extend, **`sb`** is for store.
- **`lh, lhu, sh`** does same thing for 2bytes.

Null-terminated string copy example

```
void strcpy (char x[], char y[]) { // in $a0, \$a1
    int i = 0; // in $s0
    while ((x[i]=y[i])!='\0') i += 1; }
```

```
strcpy:
    addi $sp, $sp, -4      # adjust stack for 1 item
    sw   $s0, 0($sp)       # save $s0
    add  $s0, $zero, $zero # i = 0
L1: add  $t1, $s0, $a1     # addr of y[i] in $t1
    lbu  $t2, 0($t1)       # $t2 = y[i]
    add  $t3, $s0, $a0     # addr of x[i] in $t3
    sb   $t2, 0($t3)       # x[i] = y[i]
    beq  $t2, $zero, L2    # exit loop if y[i] == 0
    addi $s0, $s0, 1       # i = i + 1
    j    L1                # next iteration of loop
L2: lw   $s0, 0($sp)       # restore saved $s0
    addi $sp, $sp, 4       # pop 1 item from stack
    jr   $ra               # and return
```

32-bit constants. Most constants are small → 16-bit immediate is sufficient. For the occasional 32-bit constant:

`lui $s0, 61`	0000 0000 0111 1101	0000 0000 0000 0000
`ori $s0, $s0, 2304`	0000 0000 0111 1101	0000 1001 0000 0000

3.11 Addressing

3.11.1 Branch addressing

Branch instructions specify `opcode`, two registers, and target address. Most branch targets are near branch → 16 bits is sufficient. **PC-relative addressing**: target address = PC + offset × 4. PC already incremented by 4 by this time.

3.11.2 Jump addressing

Jump (`j` and `jal`) targets could be anywhere in text segment.

(Pseudo) Direct jump addressing: target address = leftmost 4 bits from PC + instruction ×4 → assumes that target is not that far.

3.12 Sort example

```c
void swap(int v[], int k) {
   int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp; }
void sort (int v[], int n) {
    int i, j;
    for (i = 0; i < n; i += 1) {
        for (j = i-1; j >= 0 && v[j] > v[j+1]; j -= 1)
            swap(v,j); }}
```

`swap` procedure is leaf. `v` in `$a0`, `k` in `$a1`, `temp` in `$t0`

```
swap: sll $t1, $a1, 2   # $t1 = k * 4
      add $t1, $a0, $t1 # $t1 = v+(k*4)
                        #   (address of v[k])
      lw $t0, 0($t1)    # $t0 (temp) = v[k]
      lw $t2, 4($t1)    # $t2 = v[k+1]
      sw $t2, 0($t1)    # v[k] = $t2 (v[k+1])
      sw $t0, 4($t1)    # v[k+1] = $t0 (temp)
      jr $ra            # return to calling routine
```

`sort` is non-leaf. `v` in `$a0`, `k` in `$a1`, `i` in `$s0`, `j` in `$s1`.

```
         move $s2, $a0           # save $a0 into $s2                 Move params
         move $s3, $a1           # save $a1 into $s3
         move $s0, $zero         # i = 0                             Outer loop
for1tst: slt  $t0, $s0, $s3      # $t0 = 0 if $s0 ≥ $s3 (i ≥ n)
         beq  $t0, $zero, exit1  # go to exit1 if $s0 ≥ $s3 (i ≥ n)
         addi $s1, $s0, –1       # j = i – 1
for2tst: slti $t0, $s1, 0        # $t0 = 1 if $s1 < 0 (j < 0)
         bne  $t0, $zero, exit2  # go to exit2 if $s1 < 0 (j < 0)
         sll  $t1, $s1, 2        # $t1 = j * 4
         add  $t2, $s2, $t1      # $t2 = v + (j * 4)                 Inner loop
         lw   $t3, 0($t2)        # $t3 = v[j]
         lw   $t4, 4($t2)        # $t4 = v[j + 1]
         slt  $t0, $t4, $t3      # $t0 = 0 if $t4 ≥ $t3
         beq  $t0, $zero, exit2  # go to exit2 if $t4 ≥ $t3
         move $a0, $s2           # 1st param of swap is v (old $a0)  Pass params & call
         move $a1, $s1           # 2nd param of swap is j
         jal  swap               # call swap procedure
         addi $s1, $s1, –1       # j –= 1                            Inner loop
         j    for2tst            # jump to test of inner loop
exit2:   addi $s0, $s0, 1        # i += 1                            Outer loop
         j    for1tst            # jump to test of outer loop
```

```
sort:    addi $sp,$sp, –20       # make room on stack for 5 registers  Saving Registers
         sw $ra, 16($sp)         # save $ra on stack
         sw $s3,12($sp)          # save $s3 on stack
         sw $s2, 8($sp)          # save $s2 on stack
         sw $s1, 4($sp)          # save $s1 on stack
         sw $s0, 0($sp)          # save $s0 on stack
         …                       # procedure body
         …
exit1:   lw $s0, 0($sp)          # restore $s0 from stack              Restoring Registers
         lw $s1, 4($sp)          # restore $s1 from stack
         lw $s2, 8($sp)          # restore $s2 from stack
         lw $s3,12($sp)          # restore $s3 from stack
         lw $ra,16($sp)          # restore $ra from stack
         addi $sp,$sp, 20        # restore stack pointer
         jr $ra                  # return to calling routine
```

3.13 Producing an object module

Assembler (or compiler) translates program into machine instructions.

3.13.1 Linking produces an executable image

1. Merges segments
2. Remove labels (determine their addresses)
3. Patch location-dependent and external references

Program can be loaded into absolute location in VM space.

3.13.2 Load a program: From image file on disk into memory

1. Read header to determine segment sizes
2. Create virtual address space
3. Copy text and initialized data into memory
   - Or set page table entries so they can be faulted in
4. Set up arguments on stack
5. Initialize registers (including `$sp, $fp, $gp`)
6. Jump to startup routine
   - Copies arguments to `$a0`, ... and calls main
   - When main returns, do exit syscall

3.14 RISC (reduced) vs. CISC (complex)

MIPS is RISC, x86 is CISC. Before RISC, people wanted to reduce *semantic gap* between HW and high-level PLs. → bad idea. Compilers are good at making fast code from simple instructions. RISC pushes the complications to compilers and software. CISC has complications in ISA. **Backward** compatibility: Instruction set only increase.

4 Processor

Microarchitecture is implementation of ISA. We will examine **two** implementations of simple subset of MISC:

- Memory reference: `lw, sw`
- Arithmetic/logical: `add, sub, and, or, slt`
- Control transfer: `beq, j`

Generic implementation:

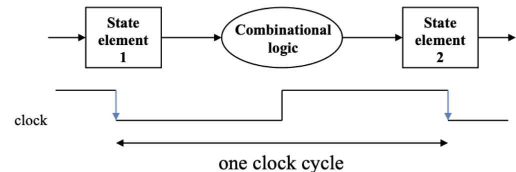
1. Use PC to supply the instruction address and **fetch** the instruction from memory (and update PC)
2. **Decode** the instruction (and read registers)
3. **Execute** the instruction

All instructions except `j` use the ALU (actual computation) after reading the registers.

4.1 Clocking

The clocking methodology defines when data in a state element is valid and stable relative to the clock

- State elements: a memory element such as a register
- Edge-triggered: all state changes occur on a clock edge



Assumes state elements are written on every clock cycle; if not, need explicit write control signal.

4.2 Datapath and control

4.2.1 Single cycle design

Fetch, decode and execute each instructions in one clock cycle

- No datapath resource can be used more than once per instruction, so some must be duplicated, *e.g.* instruction memory (IM) and data memory (DM), several adders
- Multiplexors needed at the input of shared elements with control lines to do the selection
- Write signals to control writing to register file (RF) and DM

👍 Simple and easy to understand
👎 Wasteful. Cycle time is determined by length of the longest path.

4.2.2 Instruction critical (longest) path

What is the clock cycle time assuming negligible delays for muxes, control unit, sign extend, PC access, shift left 2, wires except:

	IM	Reg Rd	ALU Op	DM	Reg Wr	Total
R	200	100	200		100	600
lw	200	100	200	200	100	**800**
sw	200	100	200	200		700
beq	200	100	200			500
j	200					200

lw is critical path

- Not feasible to vary period for different instructions
- Violates design principle – Making the common case fast
  ∵ `lw` is much rare than R-type instructions

4.2.3 How to generate control signals

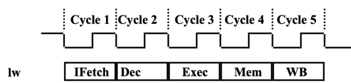
- Microcoded control
  - Control signals defined in a small memory inside the processor
  - Flexible (can be updated after the processor is manufactured)
  - Slow, used in old CISC processors
- Hardwired control
  - Generate control signal from combinatorial logics
  - Fast, but cannot be changed
  - Used in RISC (cannot make so many logics for CISC)

4.3 Pipelining (ILP)

Fetch and execute more than one instruction at a time. Under **ideal** conditions and with a large number of instructions, the speedup is ≈ to number of pipe stages. 💬 **Ideal** means we do all stages for different instructions. So CPI remains same as 1, but CC is reduced.

4.3.1 5 stages of instruction

Multiple tasks operating simultaneously using different resources.

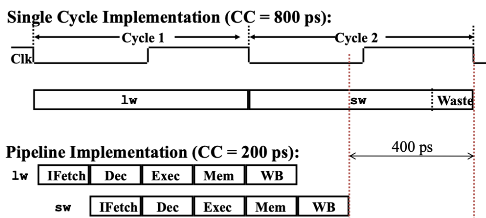


Stage	Description	Resource
IFetch (IF)	Instruction fetch and update PC	IM
Dec (ID)	Registers fetch and instruction decode	Reg
Exec (EX)	Execute or calculate memory address	ALU
Mem	Read/write the data from/to DM	DM
WB	Write the result data into RF	Reg

4.3.2 A pipelined MIPS processor

Start the next instruction before the current one has completed.

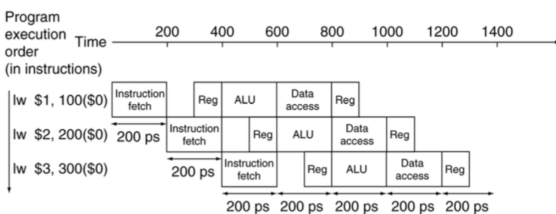
- Improves **throughput**: total amount of work done in a given time
- Instruction **latency** (execution /response time, etc) is increased.
  ∵ CC (pipeline **stage** time) is limited by the slowest stage. Some stages doesn't need to whole CC *e.g.* WB.
  ∵ Overhead to divide instruction to different stages (t_{latch})
  - For some instructions, some stages are wasted *i.e.* nothing happens.



Latency of `lw` increased (800ps → 1000ps).

4.3.3 Formalization

- Single cycle design: $t_{\text{clk}} = t_F + t_D + t_X + t_M + t_W$
- Pipelining: $t_{\text{clk}} = \max(t_F, t_D, t_X, t_M, t_W) + t_{\text{latch}}$



Reg on the right half is Dec (read), left half is WB (write). So can access RF for two different instructions.

💬 Suppose we execute 100 instructions.

- Single cycle machine: 45ns/cycle × 1 CPI × 100inst = 4500ns
- Ideal pipelined: 10ns/cycle × (1 CPI × 100 + 4 cycle drain) = 1040ns

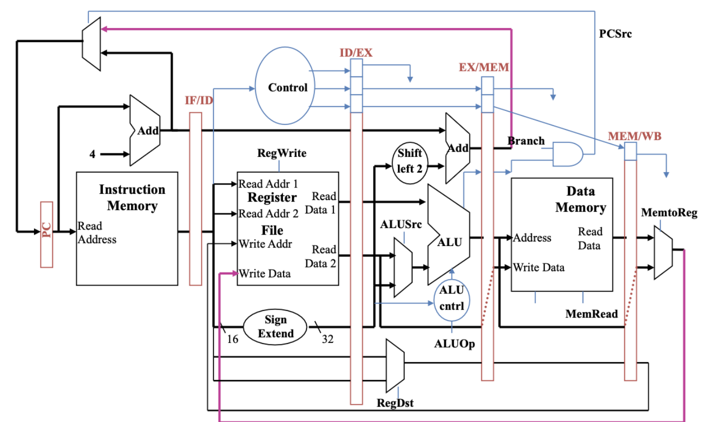
Cycle drain/pipeline fill: initial number of cycles that cannot run all stages.

4.3.4 MIPS ISA designed for pipelining

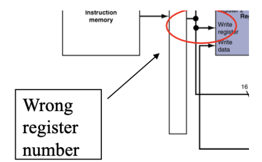
- All instructions are 32-bits → easier to fetch and decode in one cycle. ↔ x86 has 1- to 17-byte instructions.
- Few and regular instruction formats → Can decode and read registers in one step
- Load/store addressing: calculate address in Exec (3rd stage) and access memory in Mem (4th stage).
- Alignment of memory operands. You always read one word. → Memory access takes only one cycle.

4.3.5 MIPS piepline datapath

State registers between each pipeline stage to isolate them. Feed in next cycle.



- The write addr passed to all state registers for WB in future. If not, we will write in wrong register. (right)
- All control signals determined during Decode and held in the state registers.

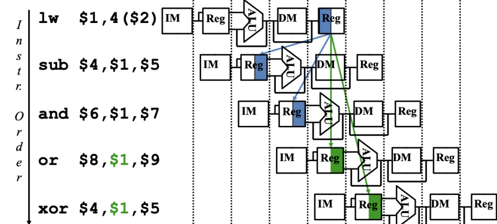


Instruction	Execution/address calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
	RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
R-format	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X

Control classified to EX/MEM/WB

4.4 Pipeline hazards

- **Structural** hazards: attempt to use the same resource by two different instructions at the same time. We already solved two:
  - We separated IM and DM (as two different caches)
  - We separated RF read and write (each consumes half cycle)
- **Data** hazards: attempt to use data before it is ready
  - An instruction's source operand(s) are produced by a prior instruction (R or `lw`, two cases) still in the pipeline



RF read of `or` in same cycle is fine, `sub`/`and` aren't. 2 cycles stall needed.

- **Control** hazards: attempt to make a decision about program control flow before (i) the condition has been evaluated (branch or not?) and (ii) the new PC target address calculated.

Can resolve hazard by waiting or bubble/stall. But impacts CPI.

4.5 Data forwarding/bypassing

Take the result from the earliest point that it exists in any of the pipeline state registers and forward it to the functional units that need it that cycle. Need multiplexors and the proper control.

4.5.1 EX forwarding

ALU input come from not only ID/EX but also EX/MEM and MEM/WB.

EX forward unit. Instruction order is A→B →C. A results is in MEM/WB. B is in EX/MEM. C is in ID/EX. Control for EX of C is...

```
if (EX/MEM.RegWrite // B will write sth in future
    and (EX/MEM.RegisterRd != 0) // B will write in rd
    and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
        ForwardA = 10
if (MEM/WB.RegWrite // A will write sth in future
    and (MEM/WB.RegisterRd != 0) // A will write in rd
    and (EX/MEM.RegisterRd != ID/EX.RegisterRs) // Both
      A, B can change Rs. Need to choose most recent one.
      So ensure B is not writing.
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
        ForwardA = 01 // so we have 2-bit signal
// Exactly same for Rt, just ForwardB varaible
```

Achieve CPI=1 even in the presence of data dependencies.

4.6 Load-use data hazards

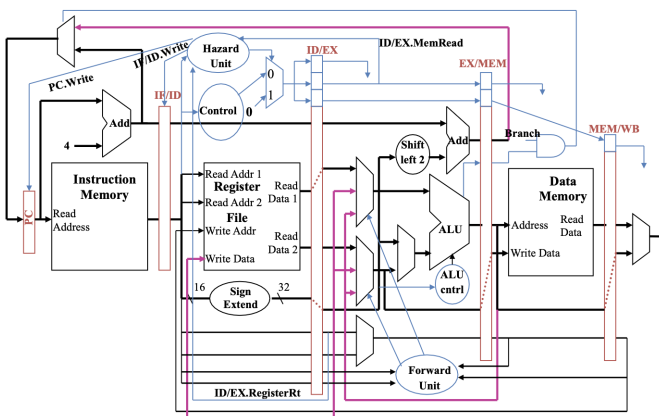
Forwarding can solve series of R-format instructions on same register, but not immediate usage after `lw`. → Need a hazard detection unit in the ID stage that inserts a stall between load and its use.

ID hazard detection unit

```
if (ID/Ex.MemRead
    and ((ID/EX.RegisterRt = IF/ID.RegisterRs)
    or (ID/EX.RegisterRt = IF/ID.RegisterRt)))
        stall the pipeline
```

4.6.1 Implementing stall

- Prevent instructions in IF and ID stages from progress. → Prevent PC and IF/ID register update by disabling write (control signal).
- Insert *bubble* or *noop* = set control bits in EX, MEM, and WB control fields to 0 → nothing will happen!
- Let `lw` and the instructions after it in the pipeline (before it in the code) proceed normally down the pipeline



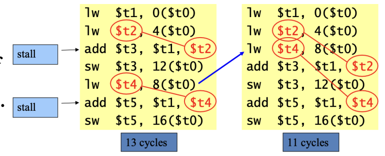
The datapath with forwarding and hazard detection unit.

4.6.2 Memory-to-memory copies

For consecutive *lwsw*, can avoid a stall by adding forwarding hardware from the MEM/WB register to the DM input. Would need to add a forward unit and a mux to the MEM stage.

4.7 Code scheduling

Reorder code to avoid use of load result in next instruction. `A=B+E;C=B+F;`



4.8 Control hazard

4.8.1 Jumps incur one stall

Jumps are unconditional. In most cases jump target is not `PC+4`. Since jump is not decoded until ID, one flush is needed. New control path (and `IF.Flush` bit) from control unit to new mux between IM and IF/ID register. Choose between fetched instruction and noop. Fortunately, jumps are very infrequent.

4.8.2 Moving branch decision earlier

Branching is decided after ALU, at **Mem**.

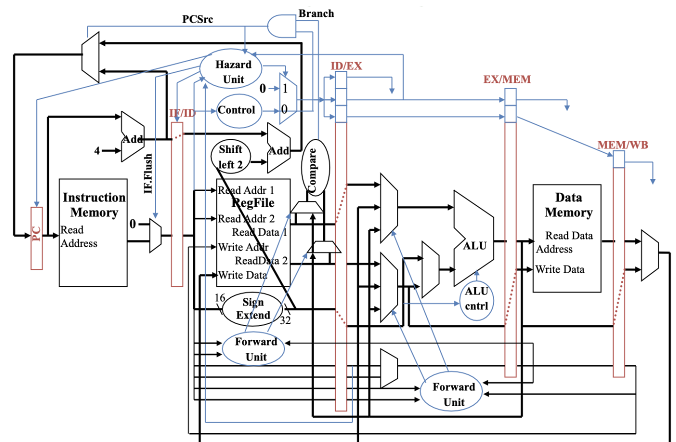
- Should flush 3 *instr*s. We have `ID.Flush, EX.Flush` bits now.
- In MIPS, state changing operations are at the end (Mem and WB) so flushed instructions haven't changed the machine state

Decision at EX = 2 stalls. Instead of getting comparison result from ALU in Mem stage, connect ALU inputs to AND gate and connect mux which is controlled by ID/EX register.

Decision at ID = 1 stall.

- Need forwarding hardwares. Register values that decides condition may be modified by preceding instructions still in pipeline.
  - MEM/WB forwarding is not needed, RF write is before read
  - EX/MEM should be forwarded. Same as EX forward unit, only `EX/MEM.RegWrite` replaced to `IDcontrol.Branch` and `ForwardA/B=10` to `ForwardC/D=1`.
  - If instruction immediately before branch produces one of the branch source operand, then stall need to be inserted in between.
- Computing branch target address can be in parallel with RF read
- Comparing registers should be done after RF read, so comparing and updating PC adds a mux, a comparator, and AND gate → increase CC, but CPI is reduced

For deeper pipelines, branch decision points can be even later, incurring more stalls. So we might want to sacrifice CC for CPI.



The datapath with ID forwarding unit

4.9 Static branch prediction

Resolve branch hazards by assuming a given outcome w/o waiting to see actual outcome.

1. Predict **not taken**: always fetch `PC+4`, flush instructions if branch should be taken
2. Predict **taken**: need to know the branch target address (BTA) = need a stall → or cache! (see below)

Falling through the loop is more common. So predict not taken works well for top loop, but not for bottom loop.

```
Loop: beq $1,$2,Out
      1st loop instr
      .
      .
      .
      last loop instr
      j  Loop
Out:  fall out instr
```

(a) Top of loop. Has `j`, → stall.

```
Loop: 1st loop instr
      2nd loop instr
      .
      .
      .
      last loop instr
      bne $1,$2,Loop
      fall out instr
```

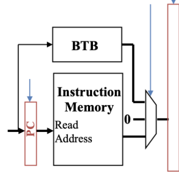
(b) Bottom of the loop

Branching structures

As the branch penalty increases (for deeper pipelines), a simple static prediction scheme will hurt performance.

3. Dynamic branch prediction with a branch prediction buffer (or branch history table (BHT)) in the IF stage
   - Addressed by the lower bits of the PC
   - Store bit: whether branch was taken last time it was executed

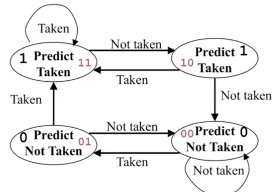
A branch target buffer (BTB) in the IF stage cache the instruction. 💡 If BTB cache BTA, IM should have two read port: one for fetching `PC+4`, one for fetching BTA. This is expensive.



4.9.1 Prediction accuracy

1-bit predictor. Assume predict bit = 0. It will be incorrect twice: first time through the loop and exiting the loop. 💬 For 10 times through the loop, and the branch is taken 9 times, accuracy is 80%.

Consider a nested loop. `TTTT NT TTTT NT...` We can improve to 90% accuracy by **2-bit predictor.** Must be wrong twice before changing preidction.



4.10 Exceptions

Exceptions (or interrupts) are just another form of control hazard.

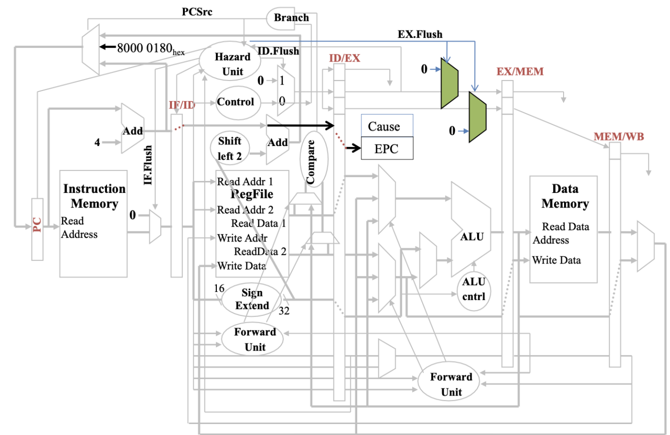
- **Interrupts**: asynchronous to program execution; external cause
  - May be handled between instructions, so can let the instructions currently active in the pipeline complete before passing control to the OS interrupt handler
  - Simply suspend and resume user program
- **Traps** (exception): synchronous to program execution; internal cause
  - The pipeline has to stop executing the offending instruction, *midstream* in the pipeline:
    * Let all prior instructions complete
    * Flush all following instructions
    * Set a register `Cause` to show the cause of the exception
    * Set a register `EPC` the address of the offending instruction
    * Jump to a prearranged address of trap handler code
  - OS looks the cause of the exception and deals with it. The offending instruction may be retried (or simulated by the OS) and the program may continue or aborted.

Exception cause	Stage	Sync?
Arithmetic overflow	EX	✓
Undefined instruction	ID	✓
TLB or page fault	IF, Mem	✓
I/O service request	any	✗
Hardware malfunction	any	✗

Multiple exceptions can occur simultaneously in a single cycle! Hardware sorts the exceptions so that the earliest instruction is the one interrupted first.

4.10.1 MIPS to handle exceptions

- Signals `CauseWrite/EPCWrite` to control writes to registers `Cause/EPC`
- Expand PC input mux; new input is wired to exception handler address



The datapath with controls for exceptions

5 Cache

Memory technology	Access time	cost/GB
Static RAM (SRAM)	0.5-2.5ns	2K-5K$
Dynamic RAM (DRAM)	50-70ns	20-75$
Magnetic disk	5-20ms	0.2-2$

	SRAM	DRAM
Used for	Cache	Main memory
Density	Low (6[1])	High (1)
Power	Higher	Lower
Content	Static (last forever)	Dynamic (refreshed regularly)

SRAM vs. DRAM. [1]Number of transistor cells.

5.1 The memory bottleneck

- Typical CPU clock rate is 2GHz (=0.5ns cycle time).
- Typical DRAM access time is 30ns ≈ 60 cycles.
- Typical main memory access is 100ns (200 cycles): DRAM (60), precharge (20), chip crossings (60, overhead (60).
- Average instruction references are 1 instruction word, 0.3 data word.

Memory delay is mostly communication time. Read/write a bit is fast. It takes time to select right bit and route the data to/from bit. This problem gets worse (processor-memory performance gap). CPUs get faster, memories get bigger.

5.2 Memory hierarchy

Large memories are slow and fast memories are small. 💡 Take advantage of the **principle of locality** to present the user with as much memory as is available in the cheapest technology at the speed offered by the fastest technology.

- Registers ↔ memory: by compiler
- Cache ↔ main memory: by cache controller hardware
- Main memory ↔ disks: by OS (VM), V-to-P mapping (TLB), and programmer (files)