Chapter 2

Instructions: Language of the Computer

Instruction Set

§2.1 Introduction

- The repertoire of instructions of a computer
- Different computers have different instruction sets
  - But with many aspects in common
- Early computers had very simple instruction sets
  - Simplified implementation
- Many modern computers also have simple instruction sets

Chapter 2 — Instructions: Language of the Computer — ‹#›

Observations

- Memory widths are much larger than the pioneering 8-bit data bus.
  - Typical memory data bus widths: 32 -128 bits
- With modern technology, CPUs can contain large numbers of registers
- Register to Register Operations are much faster than Register-Memory and/or Memory-Memory Operations

Chapter 2 — Instructions: Language of the Computer — ‹#›

Review of the VN Cycle

- Recall that John Von Neumann contributed the concept of a Stored-Program Computer.
- Relies on Fetching instructions from memory and carrying out state changes given by the operation in the instruction.
- Distinguish
  - Data Transformations (Logic and Arithmetic)
  - Flow Control
  - Data Transfers

Chapter 2 — Instructions: Language of the Computer — ‹#›

The Von Neumann Cycle (I)

Question: What is the Von Neumann Cycle?

Answer: ???? (from ICS 51/ICS 151)

The Von Neumann Cycle (II)

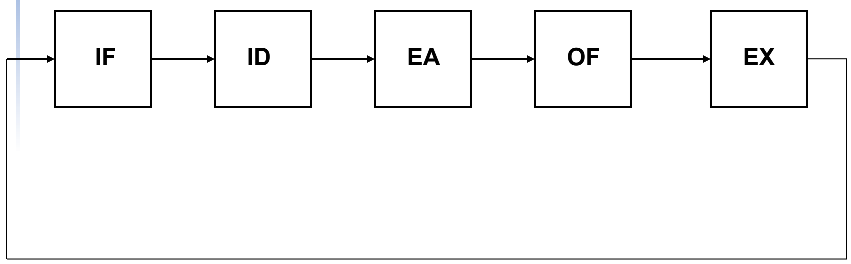
Question: What is the Von Neumann Cycle?

Answer: ???? (from ICS 51/ICS 151)

Question: How many States in the 4-state V.N. Cycle?

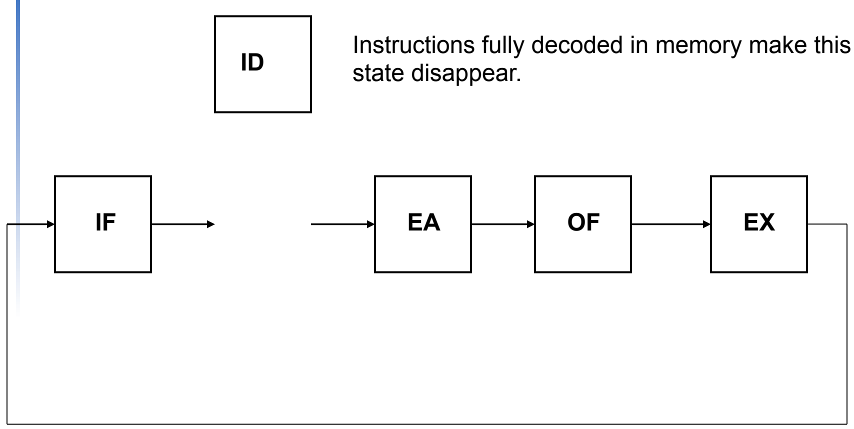
Answer: ???? (from ICS 51/ICS 151)

The Von Neumann Cycle (III)



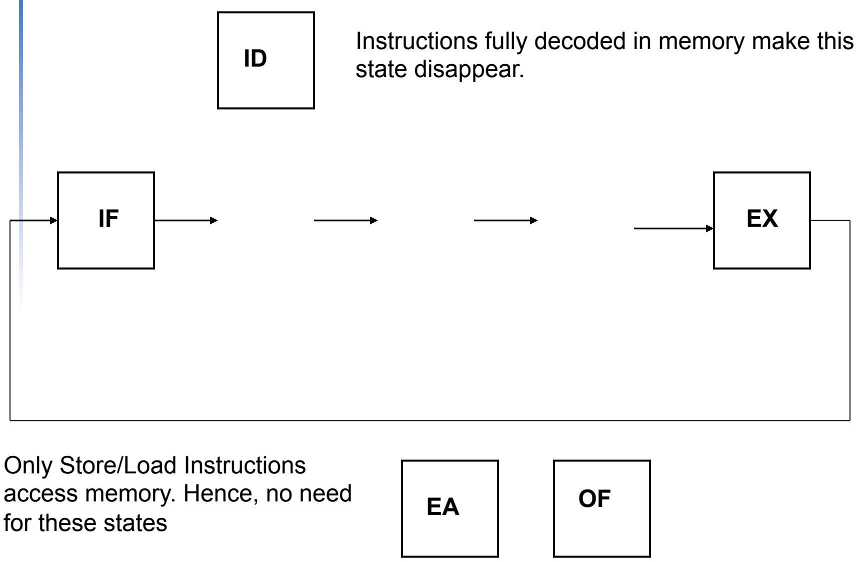
Chapter 2 — Instructions: Language of the Computer — ‹#›

The Von Neumann Cycle (III)



Chapter 2 — Instructions: Language of the Computer — ‹#›

The Von Neumann Cycle (III)



Chapter 2 — Instructions: Language of the Computer — ‹#›

The MIPS Instruction Set

- Used as the example throughout the book
- Stanford MIPS commercialized by MIPS Technologies (www.mips.com)
- Large share of embedded core market
  - Applications in consumer electronics, network/storage equipment, cameras, printers, …
- Typical of many modern ISAs
  - See MIPS Reference Data tear-out card, and Appendixes B and E

Chapter 2 — Instructions: Language of the Computer — ‹#›

Overview of MIPS

- simple instructions all 32 bits wide
- very structured, no unnecessary baggage
- only three instruction formats

R	op	rs	rt	rd	shamt	funct
I	op	rs	rt	16 bit address		
J	op	26 bit address				

- rely on compiler to achieve performance
  — what are the compiler's goals?
- help compiler where we can

Chapter 2 — Instructions: Language of the Computer — ‹#›

Three Instruction Formats

R	op	rs	rt	rd	shamt	funct
I	op	rs	rt	16 bit address		
J	op	26 bit address				

Chapter 2 — Instructions: Language of the Computer — ‹#›

Five Instruction Types

- R-type – R format
  - rd <= rs <func> rt
- I-type Conditional Branch
  - If (rs – rt) branch to PC + (sign_extend[(16-bit offset) << 2])
- I-type Memory access
  - Load: rs <= M[rt + sign_extend(16-bit offset)]
  - Store: M[rt + sign_extend(16-bit offset)] <= rs
- I-type Immediate
  - Load high/low (rs) 16-bit immediate data
- J-type Unconditional Jump
  - PC <= PC + (sign_extend[(26-bit offset) << 2])

MIPS R-format Instructions

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- Instruction fields
  - op: operation code (opcode)
  - rs: first source register number
  - rt: second source register number
  - rd: destination register number
  - shamt: shift amount (00000 for now)
  - funct: function code (extends opcode)

Arithmetic Operations

§2.2 Operations of the Computer Hardware

- Add and subtract, three operands
  - Two sources and one destination

```
add a, b, c  # a gets b + c
```

- All arithmetic operations have this form
- *Design Principle 1:* Simplicity favours regularity
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

Chapter 2 — Instructions: Language of the Computer — ‹#›

Register Operands

§2.3 Operands of the Computer Hardware

- Arithmetic instructions use register operands
- MIPS has a 32 × 32-bit register file
  - Use for frequently accessed data
  - Numbered 0 to 31
  - 32-bit data called a "word"
- *Design Principle 2:* Smaller is faster
  - c.f. main memory: millions of locations

Chapter 2 — Instructions: Language of the Computer — ‹#›

Memory Operands

- Main memory used for composite data
  - Arrays, structures, dynamic data
- To apply arithmetic operations
  - Load values from memory into registers
  - Store result from register to memory
- Memory is 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
  - *c.f.* Little Endian: least-significant byte at least address

Registers vs. Memory

- Registers are faster to access than memory
- Operating on memory data requires loads and stores
  - More instructions to be executed
- Compiler must use registers for variables as much as possible
  - Only spill to memory for less frequently used variables
  - Register optimization is important!

Immediate Operands

- Constant data specified in an instruction

  `addi $s3, $s3, 4`
- No subtract immediate instruction
  - Just use a negative constant

    `addi $s2, $s1, -1`
- *Design Principle 3:* Make the common case fast
  - Small constants are common
  - Immediate operand avoids a load instruction

The Constant Zero

- MIPS register 0 ($zero) is the constant 0
  - Cannot be overwritten
- Useful for common operations
  - E.g., move between registers

    `add $t2, $s1, $zero`

MIPS I-format Instructions

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

- Immediate arithmetic and load/store instructions
  - rt: destination or source register number
  - Constant: -2^{15} to $+2^{15} - 1$
  - Address: offset added to base address in rs
- *Design Principle 4:* Good design demands good compromises (good for US Congress ! ☺)
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible

Five Instruction Types

- R-type – R format
  - rd <= rs <func> rt
- I-type Conditional Branch
  - If (rs – rt) branch to PC + (sign_extend[(16-bit offset) << 2])
- I-type Memory access
  - Load: rs <= M[rt + sign_extend(16-bit offset)]
  - Store: M[rt + sign_extend(16-bit offset)] <= rs
- I-type Immediate
  - Load high/low (rs) 16-bit immediate data
- J-type Unconditional Jump
  - PC <= PC + (sign_extend[(26-bit offset) << 2])

Jump Addressing

- Jump (`j` and `jal`) targets could be anywhere in text segment
  - Encode full address in instruction

op	address
6 bits	26 bits

- PC-Direct jump addressing
  - Target address = $PC_{31\ldots28}$: (address × 4)
- PC-Relative jump addressing
  - Target address=PC+(signextend(26bit)<<2)

Five Instruction Types

- R-type – R format
  - rd <= rs <func> rt
- I-type Conditional Branch
  - If (rs – rt) branch to PC + (sign_extend[(16-bit offset) << 2])
- I-type Memory access
  - Load: rs <= M[rt + sign_extend(16-bit offset)]
  - Store: M[rt + sign_extend(16-bit offset)] <= rs
- I-type Immediate
  - Load high/low (rs) 16-bit immediate data
- J-type Unconditional Jump
  - PC <= PC + (sign_extend[(26-bit offset) << 2])

Target Addressing Example

- Loop code from earlier example
  - Assume Loop at location 80000

`Loop: sll  $t1, $s3, 2`	80000	0	0	19	9	4	0
`add  $t1, $t1, $s6`	80004	0	9	22	9	0	32
`lw   $t0, 0($t1)`	80008	35	9	8	0		
`bne  $t0, $s5, Exit`	80012	5	8	21	2		
`addi $s3, $s3, 1`	80016	8	19	19	1		
`j    Loop`	80020	2	20000				
`Exit: …`	80024						

Chapter 2 — Instructions: Language of the Computer — ‹#›

Branching Far Away

- Recall I-type Conditional Branch
  - If (rs – rt) branch to PC + (sign_extend[(16-bit offset) << 2])

Chapter 2 — Instructions: Language of the Computer — ‹#›

Branching Far Away

- If branch target is too far to encode with 16-bit offset, assembler rewrites the code
- Example

```
        beq $s0,$s1, L1
              ↓
        bne $s0,$s1, L2
        j L1
    L2: …
```