CS2100 COMPUTER ORGANISATION

http://www.comp.nus.edu.sg/~cs2100/

Lecture #10

Instruction Set Architecture
(ISA)

NUS National University of Singapore | School of Computing

Questions?

Ask at
https://sets.netlify.app/module/676ca3a07d7f5ffc1741dc65

OR

Scan and ask your questions here!
(May be obscured in some slides)

Lecture #10: Instruction Set Architecture

1. Overview
2. RISC vs CISC: The Famous Battle
3. The 5 Concepts in ISA Design
   - 3.1 Concept #1: Data Storage
   - 3.2 Concept #2: Memory and Addressing Mode
   - 3.3 Concept #3: Operations in Instruction Set
   - 3.4 Concept #4: Instruction Formats
   - 3.5 Concept #5: Encoding the Instruction Set

1. Overview

- We have studied MIPS but it is only one example
  - There are many other assembly languages with different characteristics
- This lecture gives a more general view on the design of Instruction Set Architecture (ISA)
- Use your understanding of MIPS and explore other possibilities/alternatives

2. RISC vs CISC: The Famous Battle

- Two major design philosophies for ISA:
- **C**omplex **I**nstruction **S**et **C**omputer (**CISC**)
  - Example: x86-32 (IA32)
  - Single instruction performs complex operation
    - VAX architecture had an instruction to multiply polynomials
  - Smaller program size as memory was premium
  - Complex implementation, no room for hardware optimization
- **R**educed **I**nstruction **S**et **C**omputer (**RISC**)
  - Example: **MIPS**, ARM
  - Keep the instruction set small and simple, makes it easier to build/optimise hardware
  - Burden on software to combine simpler operations to implement high-level language statements

3. The 5 Concepts in ISA Design

1. Data Storage
2. Memory Addressing Modes
3. Operations in the Instruction Set
4. Instruction Formats
5. Encoding the Instruction Set

3.1 Concept #1: Data Storage

- Storage Architecture
- General Purpose Register Architecture

Concept #1: Data Storage

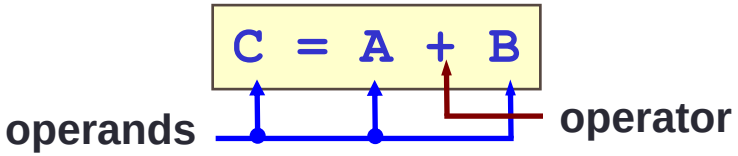
Concept #2: Memory Addressing Modes

Concept #3: Operations in the Instruction Set

Concept #4: Instruction Formats

Concept #5: Encoding the Instruction Set

3.1 Storage Architecture: Definition



Operands may be implicit or explicit.

- von Neumann Architecture:
  - Data (operands) are stored in memory
- For a processor, **storage architecture** concerns with:
  - Where do we store the operands so that the computation can be performed?
  - Where do we store the computation result afterwards?
  - How do we specify the operands?
- Major storage architectures … (next slide)

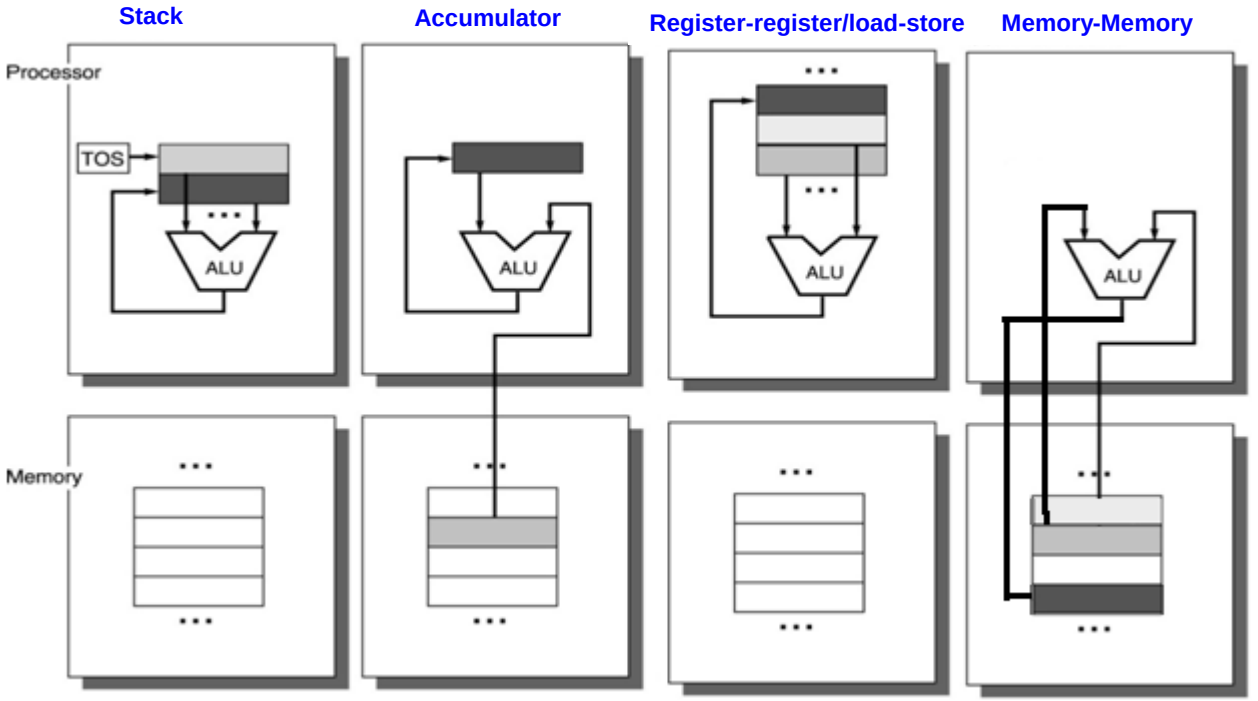
3.1 Storage Architecture: Common Design

- **Stack architecture:**
  - Operands are implicitly on top of the stack.
- **Accumulator architecture:**
  - One operand is implicitly in the accumulator (a special register). Examples: IBM 701, DEC PDP-8.
- **General-purpose register architecture:**
  - Only explicit operands.
  - **Register-memory architecture** (one operand in memory). Examples: Motorola 68000, Intel 80386.
  - **Register-register (or load-store) architecture.**
    Examples: MIPS, DEC Alpha.
- **Memory-memory architecture:**
  - All operands in memory. Example: DEC VAX.

3.1 Storage Architecture: Example

Stack	Accumulator	Register (load-store)	Memory-Memory
`Push A`	`Load A`	`Load R1,A`	`Add C, A, B`
`Push B`	`Add B`	`Load R2,B`	
`Add`	`Store C`	`Add R3,R1,R2`	
`Pop C`		`Store R3,C`	

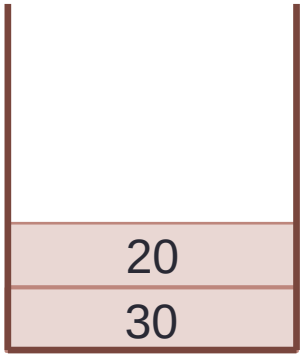
C = A+B



3.1 Storage Architecture: *Animation*

Stack	Accumulator	Register (load-store)	Memory-Memory
Push A			
Push B			
Add			
Pop C			

```
Push A
Push B
Add
Pop  C
```



Stack

...	...
10	A
20	B
30	C
...	...

Memory Variable Name

This is information remembered by the compiler.

3.1 Storage Architecture: *Animation*

Stack	Accumulator	Register (load-store)	Memory-Memory
	`Load A`		
	`Add B`		
	`Store C`		

```
Load  A
Add   B
Store C
```

	...	...
	10	A
	20	B
	30	C
30	...	...
`Accumulator`	**`Memory`**	**`Variable Name`**

This is a special register!

This is information remembered by the compiler.

3.1 Storage Architecture: *Animation*

Stack	Accumulator	Register (load-store)	Memory-Memory
		`Load R1,A`	
		`Load R2,B`	
		`Add R3,R1,R2`	
		`Store R3,C`	

```
Load R1,A
Load R2,B
Add R3,R1,R2
Store R3,C
```

...	...
10	R1
20	R2
30	R3
...	...

Registers

These are general purpose registers

...	...
10	A
20	B
30	C
...	...

Memory

Variable Name

This is information remembered by the compiler.

3.1 Storage Architecture: *Animation*

Stack	Accumulator	Register (load-store)	Memory-Memory
			`Add C, A, B`

`Add C, A, B`

...	...
10	A
20	B
30	C
...	...
`Memory`	**`Variable Name`**

This is information remembered by the compiler.

3.1 Storage Architecture: GPR Architecture

- For modern processors:
  - **General-Purpose Register** (GPR) is the most common choice for storage design
  - **RISC** computers typically use **Register-Register (Load/Store)** design
    - E.g. MIPS, ARM
  - **CISC** computers use a mixture of Register-Register and Register-Memory
    - E.g. IA32

3.2 Concept #2: Memory Addressing Mode

- Memory Locations and Addresses
- Addressing Modes

Concept #1: Data Storage

Concept #2: Memory Addressing Modes

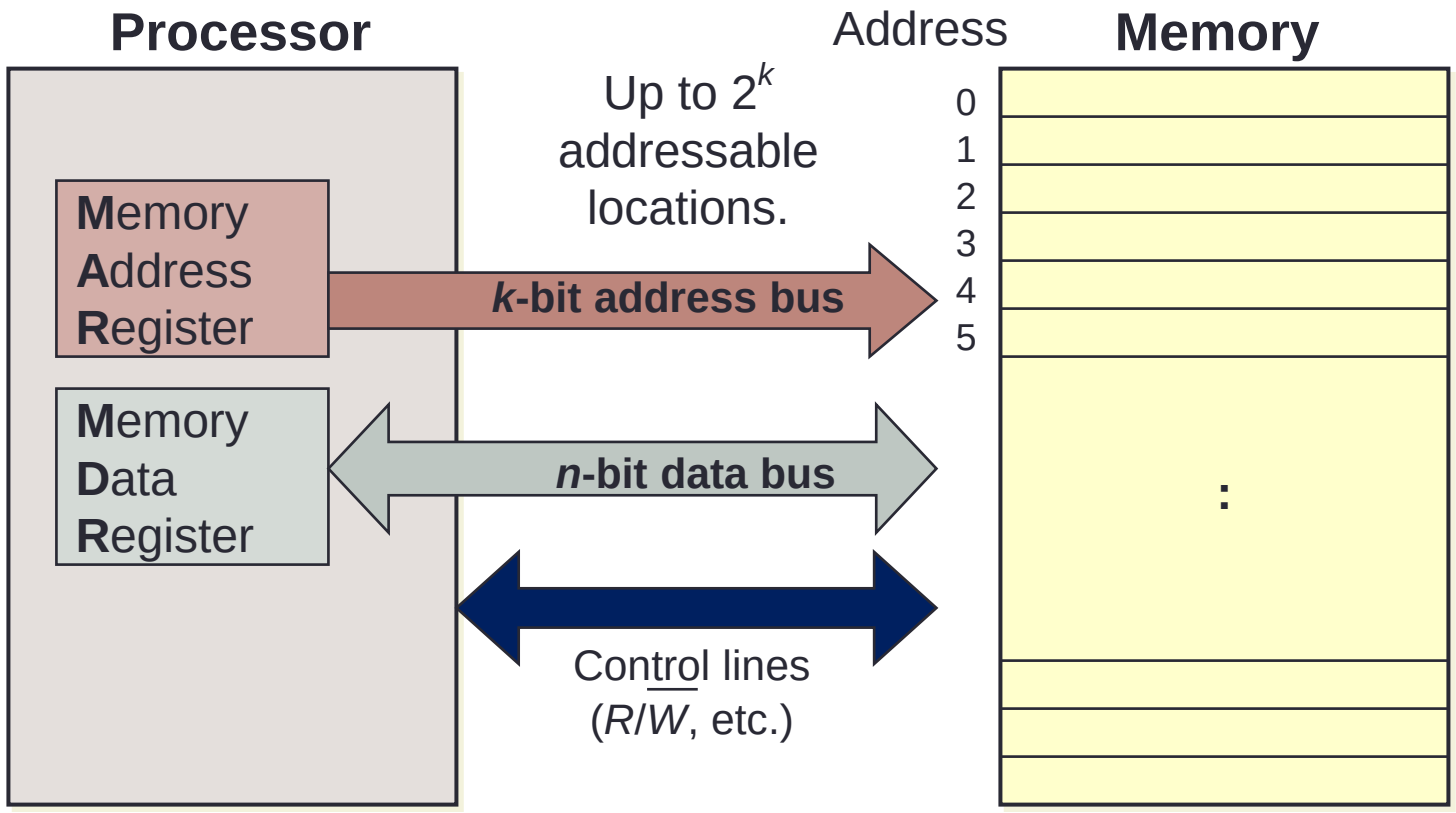
Concept #3: Operations in the Instruction Set

Concept #4: Instruction Formats

Concept #5: Encoding the Instruction Set

3.2 Memory Address and Content

- Given *k*-bit address, the address space is of size $\mathbf{2^k}$
- Each memory transfer consists of one word of ***n*** bits



3.2 Memory Content: Endianness

- **Endianness:**
  - The relative ordering of the bytes in a multiple-byte word stored in memory

Big-endian:	Little-endian:
Most significant byte stored in lowest address. **Example:** IBM 360/370, Motorola 68000, ***MIPS*** (Silicon Graphics), SPARC.	Least significant byte stored in lowest address. **Example:** Intel 80x86, DEC VAX, DEC Alpha.
Example: **0xDE AD BE EF** Stored as: 0 **DE** 1 **AD** 2 **BE** 3 **EF**	Example: **0xDE AD BE EF** Stored as: 0 **EF** 1 **BE** 2 **AD** 3 **DE**

NOTE:
The endian-ness of MIPS is actually implementation specific.

3.2 Memory Content: Endianness

- **Endianness:**
  - The online MIPS interpreter uses ***little-endian***

```
0x100   lui  $v0  , 0xDEAD
0x104   ori  $v0  , $v0  , 0xBEEF
0x108   sw   $v0  , 0($zero)
0x10C   lb   $a0  , 0($zero)
0x110   lb   $a1  , 1($zero)
0x114   lb   $a2  , 2($zero)
0x118   lb   $a3  , 3($zero)
0x11C
```

Reg	Value		
pc: $pc	0x0000011C	16	10
00: $zero	0x00000000	16	10
02: $v0	0xDEADBEEF	16	10
03: $v1	0x00000000	16	10
04: $a0	0x000000EF	16	10
05: $a1	0x000000BE	16	10
06: $a2	0x000000AD	16	10
07: $a3	0x000000DE	16	10

Addr	Value		
0x00	0xDEADBEEF	16	10
0x04	0x00000000	16	10
0x08	0x00000000	16	10
0x0C	0x00000000	16	10
0x10	0x00000000	16	10
0x14	0x00000000	16	10
0x18	0x00000000	16	10
0x1C	0x00000000	16	10

3.2 Addressing Modes

- Addressing Mode:
  - Ways to specify an operand in an assembly language
- In MIPS, there are only 3 addressing modes:
  - **Register:**
    - Operand is in a register (eg: `add $t1, $t2, $t3`)
  - **Immediate:**
    - Operand is specified in the instruction directly (eg: `addi $t1, $t2, 98`)
  - **Displacement:**
    - Operand is in memory with address calculated as Base + Offset (eg: `lw $t1, 20($t2)`)

3.2 Addressing Modes: Others

Addressing mode	*Example*	*Meaning*
Register	Add R4,R3	R4 ← R4+R3
Immediate	Add R4,#3	R4 ← R4+3
Displacement	Add R4,100(R1)	R4 ← R4+Mem[100+R1]
Register indirect	Add R4,(R1)	R4 ← R4+Mem[R1]
Indexed / Base	Add R3,(R1+R2)	R3 ← R3+Mem[R1+R2]
Direct or absolute	Add R1,(1001)	R1 ← R1+Mem[1001]
Memory indirect	Add R1,@(R3)	R1 ← R1+Mem[Mem[R3]]
Auto-increment	Add R1,(R2)+	R1 ← R1+Mem[R2]; R2 ← R2+d
Auto-decrement	Add R1,–(R2)	R2 ← R2-d; R1 ← R1+Mem[R2]
Scaled	Add R1,100(R2)[R3]	R1 ← R1+Mem[100+R2+R3*d]

3.3 Concept #3: Operations in Instructions Set

- Standard Operations in an Instruction Set
- Frequently Used Instructions

Concept #1: Data Storage

Concept #2: Memory Addressing Modes

Concept #3: Operations in the Instruction Set

Concept #4: Instruction Formats

Concept #5: Encoding the Instruction Set

3.3 Standard Operations

Data Movement	load (from memory) store (to memory) memory-to-memory move register-to-register move input (from I/O device) output (to I/O device) push, pop (to/from stack)
Arithmetic	integer (binary + decimal) or FP add, subtract, multiply, divide
Shift	shift left/right, rotate left/right
Logical	not, and, or, set, clear
Control flow	Jump (unconditional), Branch (conditional)
Subroutine Linkage	call, return
Interrupt	trap, return
Synchronisation	test & set (atomic r-m-w)
String	search, move, compare
Graphics	pixel and vertex operations, compression/decompression

NOTE:
Synchronisation is used for multi-thread or multi-core operations.
Graphics now common in x86 (*e.g., intel iris, etc*).

3.3 Frequently Used Instructions

Rank	Integer Instructions	Average %
1	Load	**22%**
2	Conditional Branch	**20%**
3	Compare	**16%**
4	Store	**12%**
5	Add	8%
6	Bitwise AND	6%
7	Sub	5%
8	Move register to register	4%
9	Procedure call	1%
10	Return	1%
	Total	**96%**

Make these instructions fast!
Amdahl's law – make the common cases fast!

NOTE:
To briefly see the benefit, consider that we managed to decrease the time needed to compute the first 4 operations by 50% (i.e., time slashed by 2) at the expense that the rest are slower (e.g., time increase by 50%).
Then if the total time originally:
T = (0.7 * t) + (0.3 * t)
After the improvement, the total time will be:
T = (0.35 * t) + (0.45 t)
T = (0.80 * t)
Which is still an improvement.

In practice, typically there is no (*or only slight*) increase in the rest when we made improvement to some.

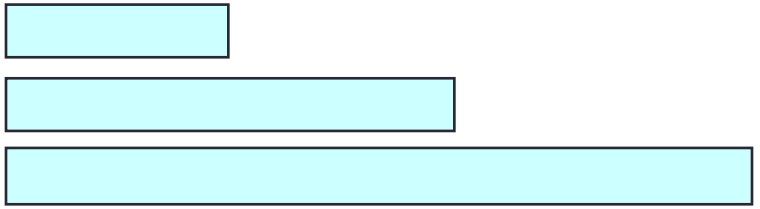
3.4 Concept #4: Instruction Formats

- Instruction Length
- Instruction Fields
  - Type and Size of Operands

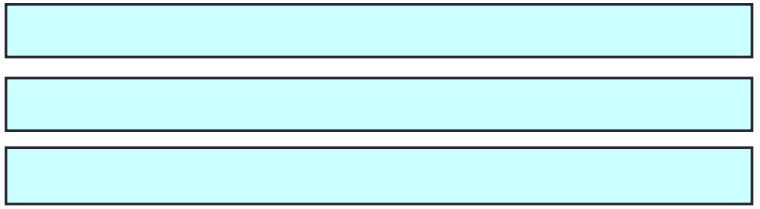
Concept #1: Data Storage
Concept #2: Memory Addressing Modes
Concept #3: Operations in the Instruction Set
Concept #4: Instruction Formats
Concept #5: Encoding the Instruction Set

3.4 Instruction Length

- **Variable-length** instructions.
  - Intel 80x86: Instructions vary from 1 to 17 bytes long.
  - Digital VAX: Instructions vary from 1 to 54 bytes long.
  - Require multi-step fetch and decode.
  - Allow for a more flexible (but complex) and compact instruction set.



- **Fixed-length** instructions.
  - Used in most RISC (Reduced Instruction Set Computers)
  - MIPS, PowerPC: Instructions are 4 bytes long.
  - Allow for easy fetch and decode.
  - Simplify pipelining and parallelism.
  - Instruction bits are scarce.



- **Hybrid** instructions: a mix of variable- and fixed-length instructions.

3.4 Instruction Fields

- An instruction consists of
  - **opcode**: unique code to specify the desired operation
  - **operands:** zero or more additional information needed for the operation
- The operation designates the type and size of the operands
  - **Typical type and size:** Character (8 bits), half-word (eg: 16 bits), word (eg: 32 bits), single-precision floating point (eg: 1 word), double-precision floating point (eg: 2 words).
- Expectations from any new 32-bit architecture:
  - Support for 8-, 16- and 32-bit integer and 32-bit and 64-bit floating point operations. A 64-bit architecture would need to support 64-bit integers as well.

3.5 Concept #5: Encoding the Instruction Set

- Instruction Encoding
- Encoding for Fixed-Length Instructions

Concept #1: Data Storage
Concept #2: Memory Addressing Modes
Concept #3: Operations in the Instruction Set
Concept #4: Instruction Formats
Concept #5: Encoding the Instruction Set

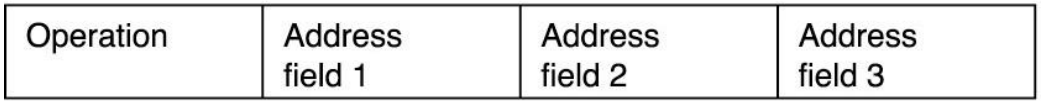
3.5 Instruction Encoding: Overview

- How are instructions represented in binary format for execution by the processor?
- Issues:
  - Code size, speed/performance, design complexity.
- Things to be decided:
  - Number of registers
  - Number of addressing modes
  - Number of operands in an instruction
- The different competing forces:
  - Have many registers and addressing modes
  - Reduce code size
  - Have instruction length that is easy to handle (fixed-length instructions are easier to handle)

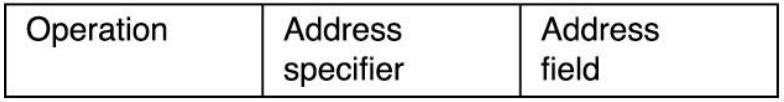
3.5 Encoding Choices

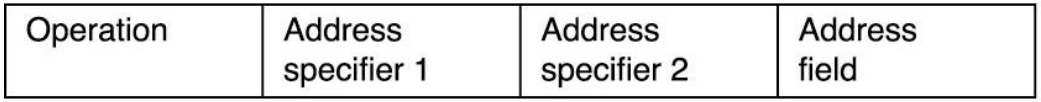
- Three encoding choices: variable, fixed, hybrid.

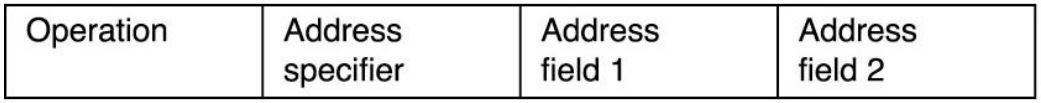
(a) Variable (e.g., VAX, Intel 80x86)



(b) Fixed (e.g., Alpha, ARM, MIPS, PowerPC, SPARC, SuperH)







(c) Hybrid (e.g., IBM 360/70, MIPS16, Thumb, TI TMS320C54x)

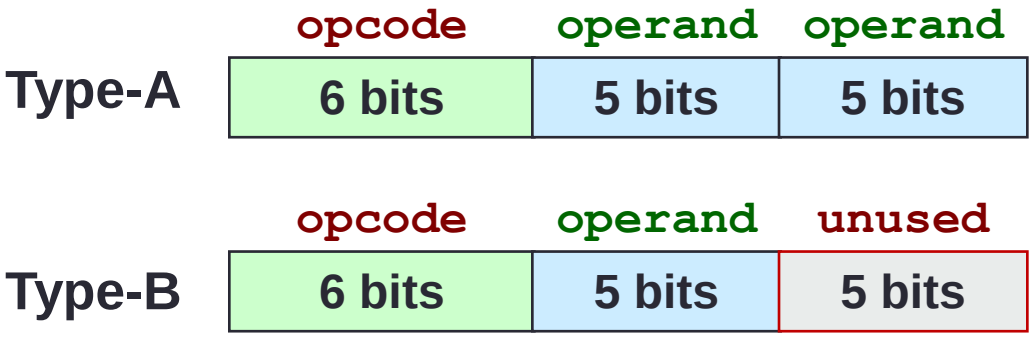
3.5 Fixed Length Instructions: Encoding (1/4)

- Fixed length instruction presents a much more interesting challenge:
  - Q: How to fit multiple sets of instruction types into same (limited) number of bits?
  - A: Work with the most constrained instruction types first
- **Expanding Opcode** scheme:
  - The opcode has variable lengths for different instructions.
  - A good way to maximize the instruction bits.

3.5 Fixed Length Instructions: Encoding (2/4)

- **Example:**
  - 16-bit fixed length instructions, with 2 types of instructions
  - **Type-A:** 2 operands, each operand is 5-bit
  - **Type-B:** 1 operand of 5-bit

First Attempt:
Fixed length Opcode



Problem:

- Wasted bits in Type-B instructions
- Maximum total number of instructions is 2^6 or 64.

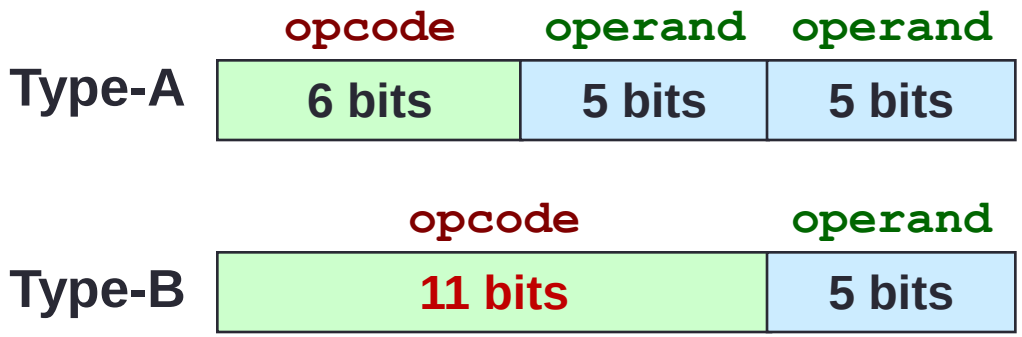
NOTE:
The opcode must be shared between type-A and type-B (e.g., 1 type-A and the rest are type-B; 1 type-B and the rest are type-A; or anything in between).
2^6 is then simply ignore the unused bits.

3.5 Fixed Length Instructions: Encoding (3/4)

- Use **expanding opcode** scheme:
  - Extend the opcode for type-B instructions to **11 bits**

➔ No wasted bits and result in a larger instruction set

Second Attempt:
Expanding Opcode

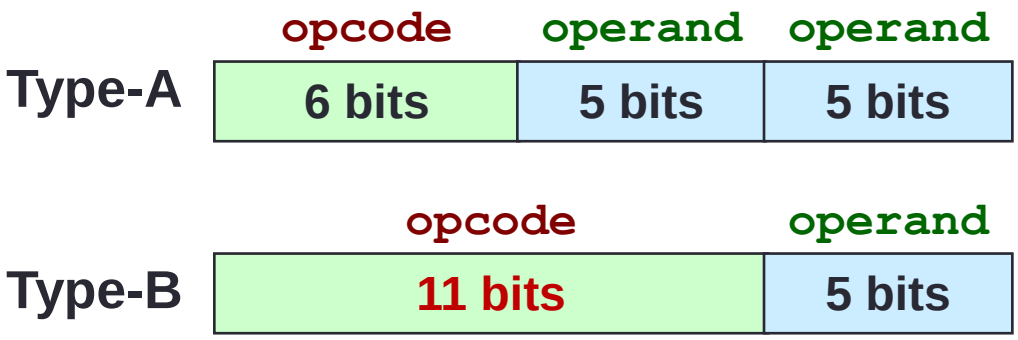


- **Questions:**
  - How do we distinguish between Type-A and Type-B?
  - How many different instructions do we really have?

Possible Answers:
1. Think about MIPS, we simply set the opcode as all 0 for R-format but use the "extended opcode" called funct to distinguish.
2. There are different ways to compute the many different instructions depending on what to maximise and/or minimise.

3.5 Fixed Length Instructions: Encoding (4/4)

- What is the maximum number of instructions?



Answer:

$1 + (2^6 - 1) \times 2^5$

$= 1 + 63 \times 32$

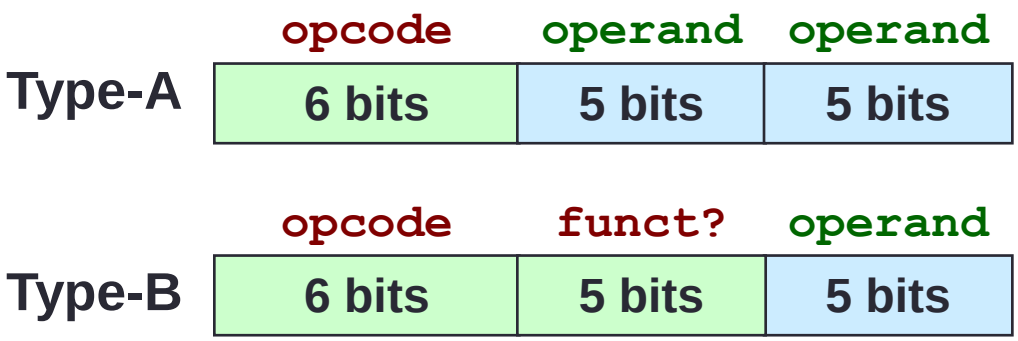
$=$ **2017**

- **Reasoning:**
  1. For every 6-bit prefix (front-part) given to Type-B, we get 2^5 unique patterns, e.g. `[111111]xxxxx`
  2. So, we should minimise Type-A instruction and give as many 6-bit prefixes as possible to Type-B

     ➔ 1 Type-A instruction, $2^6 - 1$ prefixes for Type-B

3.5 Extended Opcode Explanation

- What is the maximum number of instructions?



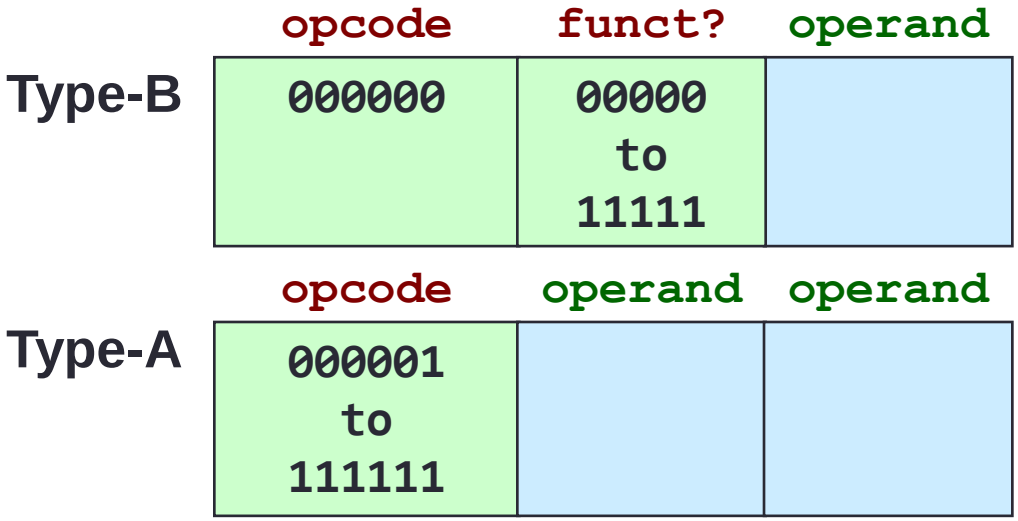
Answer:

$1 + (2^6 - 1) \times 2^5$

$= 1 + 63 \times 32$

$=$ **2017**

- **Alternative #1: Minimise Type-B** ***(use only one opcode)***



$= 1 \times 2^5 = 32$

$+$

$= 2^6 - 1$ *(1 is used for type-B)*

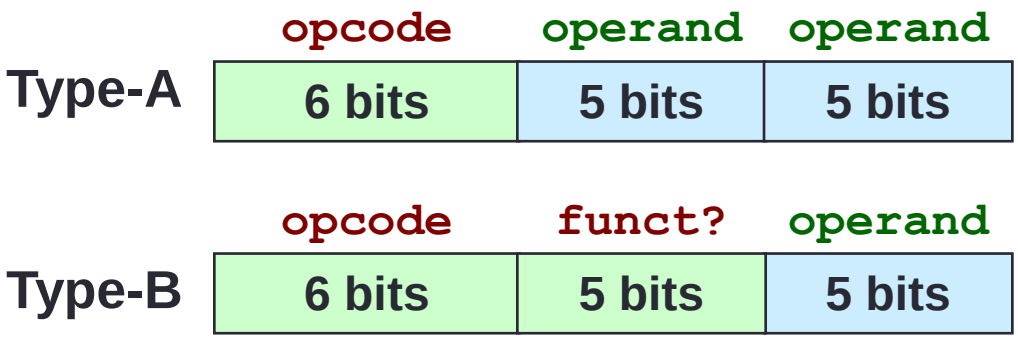
$= 63$

Total = 32 + 63 = 95

NOTE:
This actually gives the *minimum* number of instruction (assuming no unused bits and both instructions exist). Can you see why?

3.5 Extended Opcode Explanation

- What is the maximum number of instructions?



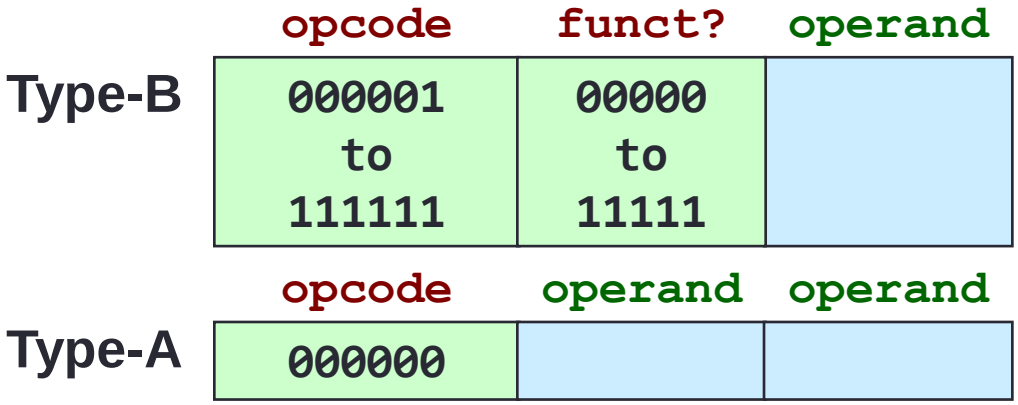
Answer:

$1 + (2^6 - 1) \times 2^5$

$= 1 + 63 \times 32$

$=$ **2017**

- **Alternative #2: Minimise Type-A** ***(use only one opcode)***



NOTE:
Clearly, alternative #2 is larger. In fact, you can check that it will give you the ***maximum*** number of instructions.

Total = 2016 + 1 = 2017

3.5 Expanding Opcode: Another Example

- Design an expanding opcode for the following to be encoded in a 36-bit instruction format. An address takes up 15 bits and a register number 3 bits.
  - 7 instructions with two addresses and one register number.
  - 500 instructions with one address and one register number.
  - 50 instructions with no address or register.

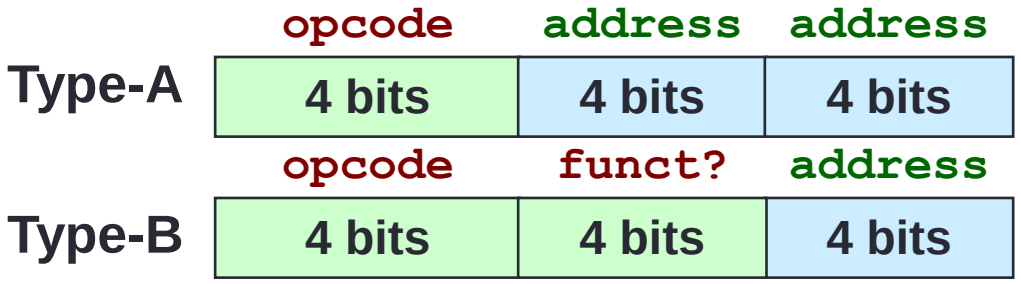
One possible answer:

3 bits	15 bits	15 bits	3 bits
000 → 110 opcode	address	address	register
111	000000 + 9 bits opcode	address	register
111	000001 : 110010 + 9 0s opcode	unused	unused

Past Midterm/Exam Questions (1/2)

- A certain machine has 12-bit instructions and 4-bit addresses. Some instructions have one address and others have two. Both types of instructions exist in the machine.

1. What is the maximum number of instructions with one address?
   a) 15
   b) 16
   c) 240
   d) 256
   e) None of the above



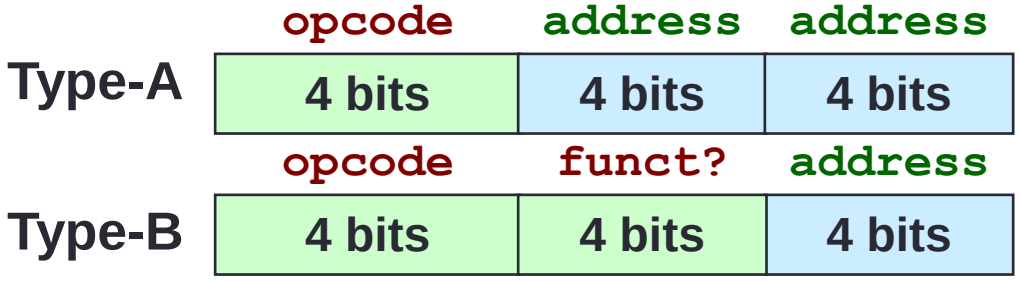
NOTE:
What is the maximum number of type-B instruction. We are only interested in type-B and not the total.
Use the same idea: *minimise type-A instruction*.
This gives us the following computation:
$(2^4 - 1) \times (2^4) = 15 \times 16 = 240$
Please discuss in forum.

Answer: (c)

Past Midterm/Exam Questions (2/2)

- A certain machine has 12-bit instructions and 4-bit addresses. Some instructions have one address and others have two. Both types of instructions exist in the machine.

2. What is the minimum total number of instructions, assuming the encoding space is completely utilised (that is, no more instructions can be accommodated)?
   a) 31
   b) 32
   c) 48
   d) 256
   e) None of the above



NOTE:
We use the alternative #1 to minimise. This gives us the following computation:
$(2^4 - 1) + (1 \times 2^4) = 15 + 16 = 31$
Please discuss in forum.

Answer: (a)

Reading

- Instructions: Language of the Computer
  - COD Chapter 2, pg 46-53, 58-71. (3rd edition)
  - COD Chapter 2, pg 74-81, 86-87, 94-104. (4th edition)

End of File