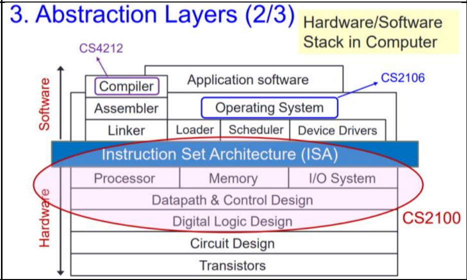
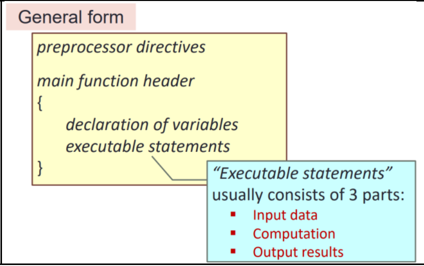
CS2100 Cheatsheet
Xu An Teh

Abstraction in Programming Langauge	
High-level Language	Level of Abstraction closer to problem domain Provides productivity and portability
Assembly Language	Textual and Symbolic Representation of Instructions
Machine Code (Object Code / Binary)	Binary Bits of Instructions and Data

Abstraction Layers in Computer	
Computer Organisation	Study of internal working, structuring and implementation of a computer system Refers to the level of abstraction above the digital logic level, but below the operating system level
Importance	Hardware and Software affect Performance Need to: build software, purchasing decision, offer “expert” advice
Hardware & Software	Algorithm determines number of source-level statements Language, compiler and architecture determine machine instructions Processor and memory determine how fast instructions are executed
Diagram	

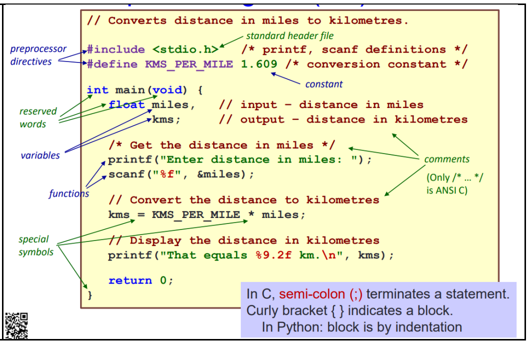
C Programming Basics	
History	A general-purpose computer programming language developed in 1972 by Dennis Ritchie (1941 – 2011) at Bell Telephone Lab for use with the UNIX operation System
Quick Review	Edit (vim), Compile (gcc), Execute (a.out)
General Form	

General form

preprocessor directives

main function header
{
declaration of variables
executable statements
}

“Executable statements” usually consists of 3 parts:
- Input data
- Computation
- Output results

Example	

```
// Converts distance in miles to kilometres.

#include <stdio.h>   /* printf, scanf definitions */
#define KMS_PER_MILE 1.609 /* conversion constant */

int main(void) {
   float miles,   // input - distance in miles
         kms;     // output - distance in kilometres

   /* Get the distance in miles */
   printf("Enter distance in miles: ");
   scanf("%f", &miles);

   // Convert the distance to kilometres
   kms = KMS_PER_MILE * miles;

   // Display the distance in kilometres
   printf("That equals %9.2f km.\n", kms);

   return 0;
}
```

In C, semi-colon (;) terminates a statement.
Curly bracket { } indicates a block.
In Python: block is by indentation

Von Neumann Architecture	
Inventor	John von Neumann (1903 - 1957)
Description	Describes a computer consisting of a CPU, Memory and I/O Devices
Central Processing Unit (CPU)	Contains Registers and a Control Unit containing an instruction register and program counter and an Arithmetic/Logic Unit (ALU)
Memory	Stores both program and data in Random-Access Memory (RAM)
I/O Devices	Input/Output Devices

Variables			
Description	Data used in a program are stored in variables		
Uses	Identified as a name (identifier), has a data type, contains a value which could be modified, and an address where they are stored in the memory or registers		
Declaration	(attributes)type name(= data);		
Data Types	int	4 bytes	[-2^31, 2^31-1]
	float	4 bytes	[]
	double	8 bytes	[]
	char	4 bytes	[\u00000, \uffffff], in ‘’ quotes
Type Strength	C is Strongly Typed: every variable to be declares with a data type Supports implicit type conversions and allows pointer values to be explicitly cast		
Placeholders	Format specifiers for values to be displayed or read		
	%c	char	printf / scanf
	%d	int	printf / scanf
	%f	float / double	printf
	%f	float	scanf
	%lf	double	scanf
	%e	float / double	printf (for scientific notation)
	Eg. %8.3f: real # (float/double) in width of 8, with 3d.p.		
Escape Sequence	\n	New Line	Next output on next line
	\t	Horizontal Tab	To next tab position in current line
	\”	Double Quote	Display a double quote
	%%	Percent	Display a percent char %

Assignment	Not just assign, but also return the value of RHS	
Arithmetic Operators		

Arithmetic Operators	Primary Expr	( ) [ ] . -> expr++ expr--	Left to Right
	Unary	*; &; + -; ++expr --expr; (typecast);	Right to Left
	Binary	* / %; + -; < > <= >=; == !=; &&; ||	Left to Right
	Ternary Opr	cond ? (exprTrue) : (exprFalse)	Right to Left
	Assignment	=, +=, -=, *=, /=, %=	Right to Left

Relational Operation	<	less than	>	more than	==	equal to
	<=	< or equal to	>=	> or equal to	!=	not equal to
Logical Opr	&&	and	||	or	!	not

Syntax of C	
#include <?.h>	Import package with .h extension stdio.h: use standard input/output functions math.h: use mathematical functions (compile with -lm in sunfire) string.h: use string functions
#define name value	Non-modifiable Assignment of the Program
sys.getsizeof(type)	Returns number of bytes for the data type
stdio.scanf/stdin.scanf	Returns input from terminal through stdin
stdio.printf/stdout.printf	Print to terminal through stdout
&(variable name)	Returns address of the memory cell where variable is stored
%p	Format specifier for pointers (hexadecimal)
type *a_ptr	Declare a pointer as a_ptr
*a_ptr	Indirection operator (dereferencing), get value of pointed variable in memory address
if (condition) {} else {}	Execute if block where condition is TRUE Execute else block where condition is FALSE
switch (variable / expression) { case value: default value:}	Execute case block whose value that matches the variable / expression input and cases below Execute default block when no case value matches variable / expression input
for (init, cond, step) {}	Execute block if condition is TRUE, with initial values init, execute step at end of loop, Loop
while (cond) {}	Execute block if condition is TRUE, Loop
do {} while (cond)	Execute do block, then loop if condition is TRUE
break	Immediately exit inner-most loop
continue	Skip to next iteration of inner-most loop

Data Representation and Number System			
Bits	The 0’s and 1’s	Nibble	4 bits (rarely used now)
Byte	8 bits	Word	Multiple of bytes
Value Range	N bits can represent up to 2^n values		
Bits required	To represent M values, ceil($\log_2$M) bits required		
Decimal Number System	Weighted-positional number system (position within the number holds a specific value) Base (aka Radix) is 10 Symbols/Digits = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }		
Binary (base 2)	Octal (base 8)	Hexadecimal (base 16)	{0, ..., 9, A,
Base/Radix R	Weights in powers of R		B, C, D, E, F }

In C,	Prefix 0 for Octal	Prefix 0x for Hexadecimal
In QTSpim,	0x for Hexadecimal	
In Verilog,	8'b for 8-bit Binary	8'h for 8-bit Binary in Hexadecimal
Least Significant Bit	Smallest value digit in weight-positional system	
Most Significant Bit	Largest value digit in weight-positional system	
Base-R to Decimal	$abc.de_R = aR^2 + bR^1 + cR^0 + dR^{-1} + eR^{-2}$	
Decimal to Base-R	For whole number: Successive Division by R till 0, remainder form the number, with first remainder is LSB and the last Remainder is MSB	
	For fractional number: Repeated Multiplication by R till 0, the carried digits / carries, form the number, with the first carry as MSB and the last carry as LSB	
Base-R to Base-R^n	For n < 0, convert each bit to |n| bits For n > 1, partition n bits then convert to 1 bit	
ASCII	1 byte, 7 bits and 1 parity bit, represents characters	
Unicode	1/2/4 bytes denoted as UTF-8, UTF-16, UTF-32	
Unsigned Numbers	Only non-negative values	
Signed Numbers	Includes positive and negative values	
Sign-and-Magnitude	1-bit sign (0: +, 1: -) and N-bit magnitude format	
1s-Complement	Given x as n-bit binary number, negated value can be found as $-x = 2^n – x – 1$ Negate: invert all bits Range for n bits: $-(2^{n-1} – 1)$ to $2^{n-1} – 1$	
	Addition: Binary addition, add carry, watch for overflow (Result if opposite sign of A and B) Subtraction: A – B = A + (-B), watch for underflow	
2s-Complement	Given x as n-bit binary number, negated value can be found as $-x = 2^n – x$ Negate: invert all bits, then add 1 Range for n bits: -2^{n-1} to $2^{n-1} – 1$	
	Addition: Binary addition, ignore carry, watch for overflow (Different 'carry in' & 'carry out' of MSB) Subtraction: A – B = A + (-B), watch for underflow	
Complement on Fractions	Same idea as Whole Numbers	
Excess-N Representation	0 starts at -N	
Fixed-Point Representation	Split # to whole and fractional parts	
Floating-Point Representation	Allow very large & very small numbers	
IEEE 754 Floating-Point Rep	Base is assumed to be 2	
Single-precision (32 bits): 1-bit sign, 8-bit exponent with bias 127 (excess-127), 23-bit mantissa Double-precision (64 bits): 1-bit sign, 11-bit exponent with bias 1023 (excess-1023), 52-bit mantissa		
mantissa	Normalised with an implicit leading bit 1 ie 110.1_2 normalised to $1.101_2 \times 2^2$, only 101 is stored in first 3 digits of mantissa field	
exponent	Exponent used to normalise, plus excess	

Pointers and Functions	
Pointers	Allow direct manipulation of memory contents Retrieve address with & operator %p format specifier
Bit manipulation operators (in C)	Allows efficient bitwise operations
Incrementing pointer	a_ptr++ causes a_ptr to increase by word length, standardise to 4 bytes
Using math library	gcc -lm flag: link to Math Library on compile
Define functions	Format: modifiers type name(params) { body }
Pass-by-value	Actual params are passed to formal params ie function takes in the value instead of the variable pointer / variable in the memory
Scope Rule (function)	Formal / Function params, local to function Vars declares in function, local to function Local params and var only accessible within functions
Function Call	When function is called, an activation record is created in the call stack and memory is allocated for the local params and vars of the function When function is done, the activation record is removed, and memory allocated for the local params and vars is released
Automatic Variables	Refers to local params and vars of a function that exist in memory only during the execution of the function
Static Variables	Exists in the memory even after the function is executed
Function prototype	Param names not required, tells compiler the existence of the function for memory allocation, functions may be properly declared later

Arrays, Strings and Structures	
Collection of Data	Grouping of data in a logical / organised representations for ease of manipulation
Arrays	Homogeneous collection of data ie fixed element types, declared with element type, name and size, elements are accessed through indexing from 0
Array Initialization	Arrays can be init at declaration Correct: int a[1] = {1}; incorrect: int a[1]; a = {1};
Array pointer	Points to the first element of the array
Array name	Is a fixed pointer, points to the first element of the array, cannot be altered
String	Array of characters with null character '\0' at the end of the array ('\0' automatically added)
fgets(str, size, stdin)	Reads size – 1 char or until newline from stdin to str, may read in newline character (to replace with '\0' if necessary)
puts(str)	Print str to stdout, terminates with newline
String functions	strlen(s): length of s strcmp(s1, s2): compare ASCII values of characters in strings s1 and s2, returns -ve int if s1 < s2, +ve int if s1 > s2, 0 if s1 == s2 (lexicographically) strncmp(s1, s2, n): compares first n chars of s2 and s1
	strcpy(s1, s2): copy s2 into s1 strncpy(s1, s2, n): copy first n chars from s2 to s1
Importance of '\0' in Strings	Ensure no illegal access of memory String functions use '\0' to as terminating char
Structures	Grouping of heterogeneous members (may be varying types)
Struct Initialization	Typedef struct { var declarations; } structName;
Access members of Structure Variable	Use dot (.) variable ie struct.member;
Passing Structure into Function	Separate copy of variable is made, local to the function and so the original variable will not be modified Use pointer if original variable should be modified
Arrow operator (->)	a_ptr->name equivalent to (*a_ptr).name Note that . (dot) has higher precedence than *, so the brackets are necessary

MIPS	
Instruction Set Architecture (ISA)	Abstraction on the interface between hardware and low-level software, includes everything programmers need to know to make the machine code work correctly Allow many implementations of varying cost and performance to run identical software

Machine Code vs Assembly Language		Machine Code	Assembly Lang
	Instruction	Binary	Human Readable
	Code	Hard & tedious	Easier to write; symbolic ver of machine code
	1000 1100 1010 000 <- ASSEMBLER <- add A, B		
		May be in hexadecimal	Pseudo-instructions

Performance	Only real instructions are counted
Syntactic Sugar	Pseudo-instructions, translation scheme from a language to the same language, for commonly used actions / instructions
Code Translation	(C code) gcc'ed to (Assembly code) assembled to (Machine code/binary code)
Components to MIPS	Processor: Performs Computation Memory Storage of code and data Bus: Bridge between Processor and Memory
Step by Step (Considerations)	Code and Data reside in memory, sent to CPU for computation in ALU through the bus Mem access is slow, so CPU temporarily store values in registers (Require loading (Mem to Reg) and storing (Reg to Mem) of data through the bus) Fast Reg-to-(Reg/Constant) Arithmetic Execution Sequence with control flow, PC Reg determines address of next instruction
Registers	Limited in number (16 - 32), 32-bits, no data type (trust the assembler that the values are correct)

MIPS Instructions	
Byte vs Word Addressing	Byte Addressing: increments by 1 byte (8-bits) Word Addressing: increments by 2^n bytes
MIPS Addressing	Follows Byte Addressing
MIPS Instructions	32-bit size, sequential
Program Counter	Special Register that keeps address of instruction being / to be executed
Magic Number 32	32 registers each 32-bits, word 32-bits, memory address 32-bits
Addressing Modes	Register, Immediate, Base/Displacement (lw, sw), PC-relative (beq/bne), Pseudo-direct (j)
Variable Mapping	Maps variables to their respective registers
Load / Store	Load: Mem to Reg, Store: Reg to Mem
Shift Left/Right Logical	sll n: multiply 2^n, srl n: divide 2^n; filled with 0's
Logical Operations	bitwise (i'th bit opr i'th bit)
NOT instruction	a NOR a = NOT a / a XOR 1's = NOT a
Label	Points to an instruction
beq / bne instruction	beq: go-to L if rs == rt, bne: go-to L if rs != rt
j instruction	Jump to Label unconditionally
Block Boundary	Due to use of first 4-bits from PC, we can only jump within a 256MB block
slt instruction	slt rd, rs, rt: rd = (rs < rt) ? 1 : 0
lui Instruction	Set upper 16 bits to imm, lower 16 bits to 0's
Loading 32-bit constant	lui 16 mbs, then ori {16{0's}, immediate}
Tip*	REFER TO GREEN CARD
Tip** Reading Modified values ie SignExtImm	n{bit}: repeat bit n times value[i]: i'th bit of value value[i:j]: i'th to j'th bits of value

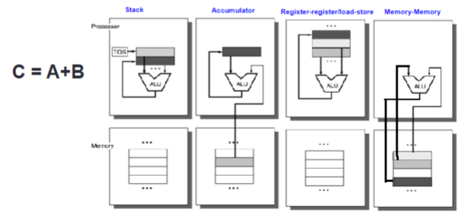
Instruction Set Architecture (ISA)	
Complex Instruction Set Computer (CISC) Eg. x86-32 (IA32)	Single instruction performs complex operation Smaller program size as memory was premium Complex implementation, no room for hardware optimization
Reduced Instruction Set Computer (RISC) Eg. MIPS, ARM	Keep instruction set small and simple, makes it easier to build/optimise hardware Burden on software to combine simpler operations to implement high-level language statements
5 Concepts in ISA Design	1. Data Storage 2. Memory Addressing Modes 3. Operations in Instruction Set 4. Instruction Formats 5. Encoding the Instruction Set
Data Storage	
Architecture	Storage Architecture General Purpose Register Architecture

Considerations	In von Neumann Architecture, Data (operands) are stored in memory Concerns: Where to store operands, results, specification
Stack Architecture	Operands are implicitly on top of the stack
Accumulator Architecture	One operand is implicitly in the accumulator (a special register) Eg. IBM 701, DEC PDP-8
General Purpose Register (GPR) Architecture	Only explicit operands Register-memory Architecture (One operand in memory) Eg. Motorola 6800, Intel 80386 Register-register (or load-store) Architecture Eg. MIPS, DEC Alpha
Memory-memory Architecture	All operands in memory Eg. DEC VAX

Visual Diagram

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`	



Real Life Application	General-Purpose Register (GPR) is most common RISC typically use Register-Register (Load/Store) CISC use a mixture of Register-Register and Register-Memory
Memory Addressing Mode	
Addressing Mode	Ways to specify an operand in an assembly language
Concept	Memory Location and Addresses, Addressing Modes
Size	Given k-bit address, address space is of size 2^k
Memory Transfer	Consists of one word of n bits
Memory Address Register	Register containing Memory Address of interest via a k-bit address bus
Memory Data Register	Register holding the value to load or store to Memory via a n-bit data bus
Control lines	Within the bus, controls data movement (to/fro)
Endianness	The relative ordering of the bytes in a multiple-byte word stored in memory
Big-endian	Most Significant Byte stored in lowest address Eg. IBM 360/370, Motorola 68000, SPARC MIPS (Silicon Graphics): implementation specific
Little-endian	Least Significant Byte stored in lowest address Eg. Intel 80x86, DEC VAX, DEC Alpha Online MIPS interpreter: little-endian
Addressing Modes in MIPS	Register: Operand in register Immediate: Operand specified in instruction directly Displacement: Operand in memory with address calculated as Base + Offset

Others

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]

Operations in Instructions Set	
Concept	Standard Operations in Instruction Set Frequently Used Instructions

Examples

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*).

Consideration (Frequency)

3.3 Frequently Used Instructions

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

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%**

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.

Instruction Formats	
Concept	Instruction Length Instruction Fields (Type & Size of Operands)
Instruction Length	Variable Length, Fixed-length, Hybrid
Variable-Length	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	Used in most RISC (Reduced Instruction Set Computer) MIPS, PowerPC: Instructions are 4 bytes long Allow for easy fetch and decode Simplify pipelining and parallelism Instruction bits are scarce
Hybrid	A mix of variable-length and fixed-length instructions
Consists of	Opcode: Unique code to specify the desired operation Operand: zero or more additional information needed for the operation
Designation	Operation designates the type and size of the operands
Typical Type and Size	(see table below)
32-bit Architecture	Support for 8-, 16- and 32-bit integer, and 32-bit and 64-bit floating point operations. 64-bit architecture would need to support 64-bit integers as well

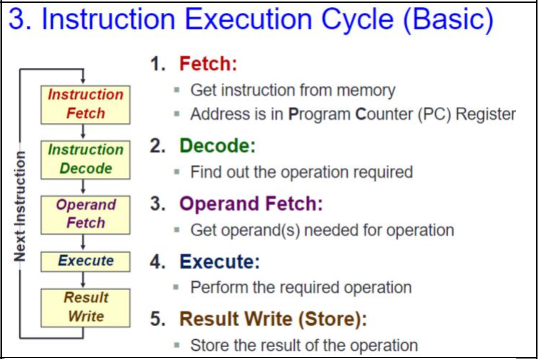
Typical Type and Size:

Character	8 bits	Half-Word	16 bits
Word	32 bits		
Single-Precision Floating Point			1 word
Double-Precision Floating Point			2 words

Encoding the Instruction Set	
Concept	Instruction Encoding Encoding for Fixed-Length Instructions
Premise	How are instructions represented in binary format for execution by the processor?
Issues	Code size, speed/performance, design complexity
Things to decide	# of registers, # of addressing modes, # of operands in an instruction
Considerations	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)
Encoding Choices	3 Encoding Choices: Variable, Fixed, Hybrid See: Instruction Formats
Expanding Opcode Scheme	Opcode with variable lengths for different instructions A good way to maximize the instruction bits
Differentiating Opcodes	Specify opcodes for specific lengths of instructions Maximum: maximise longest opcode Minimum: maximise shortest opcode

The Processor: Datapath & Control	
Datapath	Collection of components that process data
	Performs the arithmetic, logical and memory operations
Control	Tells the datapath, memory and I/O devices what to do according to program instructions
Instruction Execution Cycle	(see slide below)
MIPS Instruction Execution	(see table below)

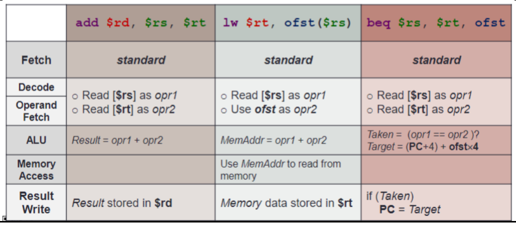
Instruction Execution Cycle:


3. Instruction Execution Cycle (Basic)

Instruction Fetch
Instruction Decode
Operand Fetch
Execute
Result Write
Next Instruction

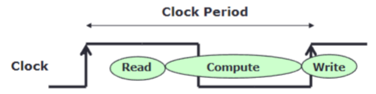
1. **Fetch:**
   - Get instruction from memory
   - Address is in **P**rogram **C**ounter (PC) Register
2. **Decode:**
   - Find out the operation required
3. **Operand Fetch:**
   - Get operand(s) needed for operation
4. **Execute:**
   - Perform the required operation
5. **Result Write (Store):**
   - Store the result of the operation

MIPS Instruction Execution:



	`add $rd, $rs, $rt`	`lw $rt, ofst($rs)`	`beq $rs, $rt, ofst`
Fetch	*standard*	*standard*	*standard*
Decode / Operand Fetch	○ Read [**$rs**] as *opr1* ○ Read [**$rt**] as *opr2*	○ Read [**$rs**] as *opr1* ○ Use ***ofst*** as *opr2*	○ Read [**$rs**] as *opr1* ○ Read [**$rt**] as *opr2*
ALU	*Result = opr1 + opr2*	*MemAddr = opr1 + opr2*	*Taken = (opr1 == opr2)?* *Target =* (**PC**+4) + **ofst×4**
Memory Access		Use *MemAddr* to read from memory	
Result Write	*Result* stored in **$rd**	*Memory* data stored in **$rt**	if (*Taken*) **PC** = *Target*

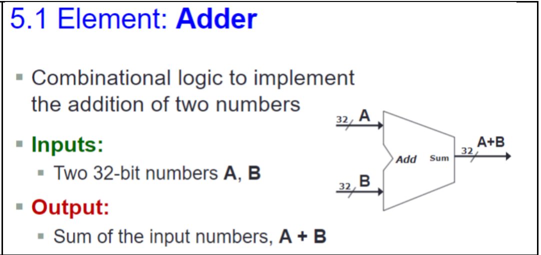
Clocking	
Clocking	Instruction Execution: - Read contents of one or more storage elements - Perform computation through some combinational logic - Write results to one or more storage elements All done within a clock period



Don't want to read a storage element when it is being written.

Single Cycle	All instructions take as much time as the slowest one
Multicycle	Execute Steps: Fetch, Decode, ALU, Mem R/W, Reg Write Each execution step takes one clock cycle, giving much shorter cycle time, ie, clock frequency is much higher Instructions take variable number of clock cycles to complete execution
Pipelining	Break instructions into execution steps, one per clock cycle Allow different instructions to be in different execution steps simultaneously

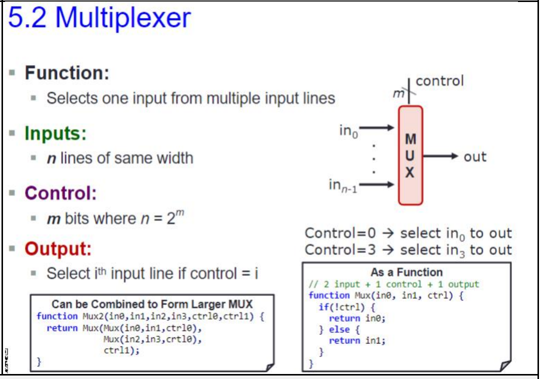
Element: Adder


5.1 Element: Adder

- Combinational logic to implement the addition of two numbers
- **Inputs:**
  - Two 32-bit numbers **A**, **B**
- **Output:**
  - Sum of the input numbers, **A + B**

32
A
32
B
Add
Sum
32
A+B

Element: Multiplexer


5.2 Multiplexer

- **Function:**
  - Selects one input from multiple input lines
- **Inputs:**
  - ***n*** lines of same width
- **Control:**
  - ***m*** bits where $n = 2^m$
- **Output:**
  - Select i^{th} input line if control = i

control
m
in_0
in_{n-1}
MUX
out

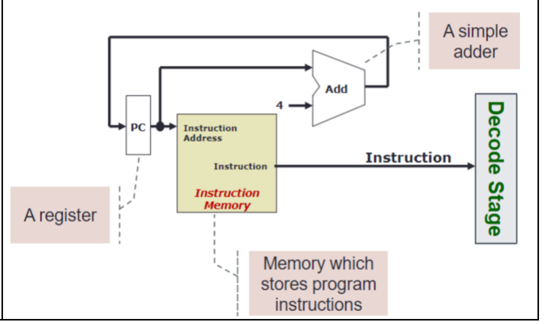
Control=0 → select in_0 to out
Control=3 → select in_3 to out

Can be Combined to Form Larger MUX
```
function Mux2(in0,in1,in2,in3,ctrl0,ctrl1) {
  return Mux(Mux(in0,in1,ctrl0),
             Mux(in2,in3,crtl0),
             ctrl1);
}
```

As a Function
```
// 2 input + 1 control + 1 output
function Mux(in0, in1, ctrl) {
  if(!ctrl) {
    return in0;
  } else {
    return in1;
  }
}
```

Datapath	
Fetch Stage	Use Program Counter (PC) to fetch the instruction form memory, PC is implemented as a special register in the processor Increment the PC by 4 to get the address of the next instruction Output to next stage (Decode): instructions to execute
Fetch Stage (Visualised)	(see diagram below)

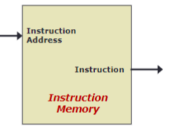
5.1 Fetch Stage: Block Diagram



Instruction Memory

5.1 Element: **Instruction Memory**

- Storage element for the instructions
  - It is a **sequential circuit** (to be covered later)
  - Has an internal state that stores information
  - Clock signal is assumed and not shown
- Supply instruction given the address
  - Given instruction address M as input, the memory outputs the content at address M
  - Conceptual diagram of the memory layout is given on the right ➔



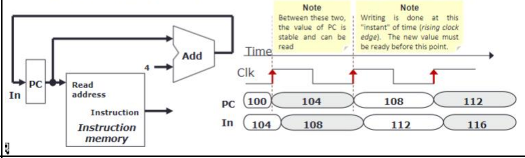
Memory

	…………
2048	add $3, $1, $2
2052	sll $4, $3, 2
2056	andi $1, $4, 0xF
……	…………

As a Function
```
function IM(addr) {
  return Mem[addr];
}
```

Magic of Clock

- **Magic of clock**:
  - PC is read during the first half of the clock period and it is updated with PC+4 at the **next rising clock edge**

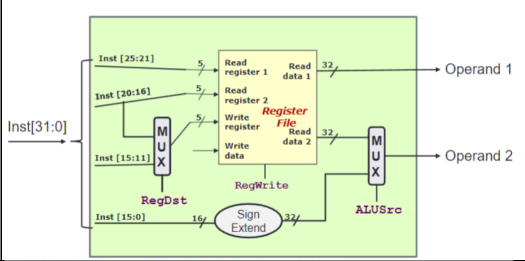

Note
Between these two, the value of PC is stable and can be read
Note
Writing is done at this "instant" of time (*rising clock edge*). The new value must be ready before this point.
Add
4
PC
In
Read address
Instruction
Instruction memory
Time
Clk
PC
100
104
108
112
In
104
108
112
116

Decode Stage

Gather data from instruction fields: opcode to determine instruction type and field lengths & data from all necessary registers
Input from previous stage (fetch): instruction to execute
Output to next stage (ALU): Operation and operands

Decode Stage (Visualised)

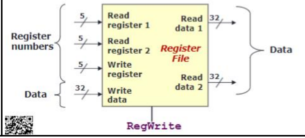
5.2 **Decode Stage**: Summary



Register File

5.2 Element: **Register File**

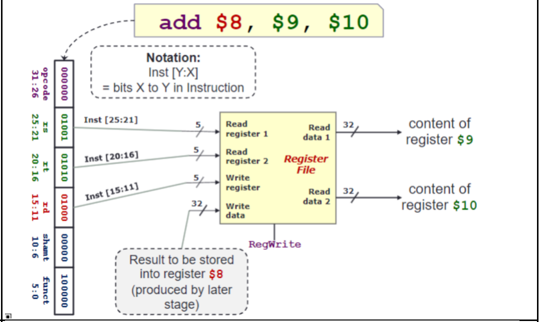
- A collection of 32 registers:
  - Each 32-bit wide; can be read/written by specifying register number
  - Read at most two registers per instruction
  - Write at most one register per instruction
- `RegWrite` is a control signal to indicate:
  - Writing of register
  - 1(True) = Write, 0 (False) = No Write



As a Function
```
// Decode Stage
function RegRead(RR1, RR2) {
  return [Reg[RR1], Reg[RR2]];
}
// Writeback Stage
function RegWrite(WR, WD, RegWrite) {
  if(RegWrite) {
    Reg[WR] = WD;
  }
}
```

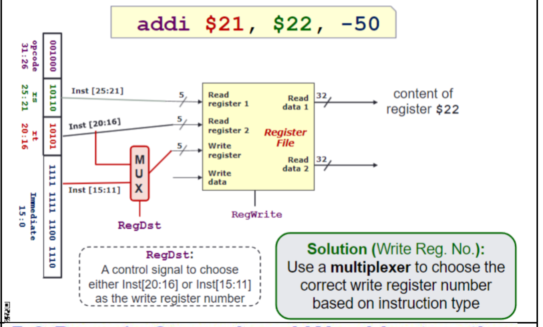
R-Format Instructions

5.2 Decode Stage: **R-Format Instruction**



I-Format Instructions

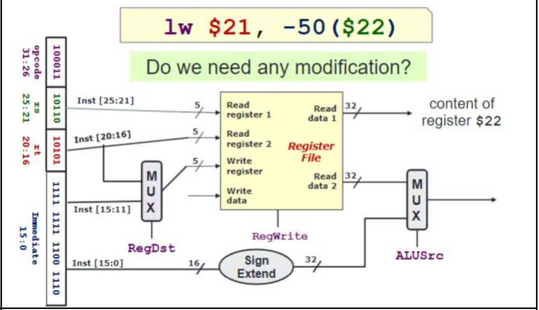
5.2 Decode Stage: **Choice in Destination**



Solution (Write Reg. No.):
Use a **multiplexer** to choose the correct write register number based on instruction type

lw / sw Instructions

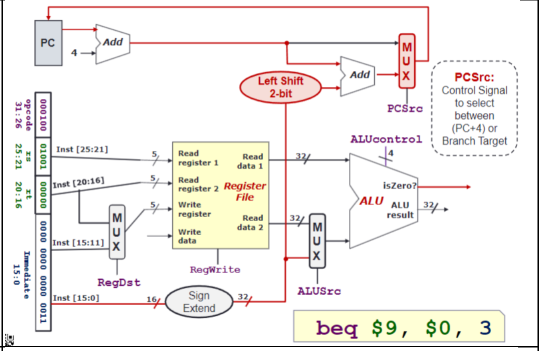
5.2 Decode Stage: **Load Word Instruction**



ALU Stage

ALU: Arithmetic-Logic Unit aka Execution stage
Performs the real work for most instructions
- Arithmetic (add, sub), Shifting (sll), Logical (and, or)
- Memory operation (lw, sw): Address calculation
- Branch operation (bne, beq): Perform register comparison and target address calculation
Input from previous stage (Decode): Operations & Operands
Output to next stage (Memory): Calculation Result

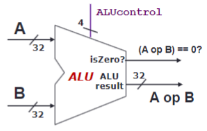
ALU Stage (Visualised)



Element: Arithmetic Logic Unit (ALU)

5.3 Element: **Arithmetic Logic Unit**

- **ALU (Arithmetic Logic Unit)**
  - Combinational logic to implement arithmetic and logical operations
- **Inputs:**
  - Two 32-bit numbers
- **Control:**
  - 4-bit to decide the particular operation
- **Outputs:**
  - Result of arithmetic/logical operation
  - A 1-bit signal to indicate whether result is zero

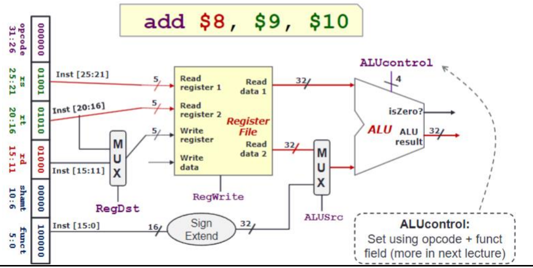


ALUcontrol	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	slt
1100	NOR

Non-Branch Instructions

5.3 ALU Stage: **Non-Branch Instructions**

- We can handle non-branch instructions easily:



Branch Instructions

Branch Outcome:
- ALU compare registers
- Output: 1-bit "isZero?" signal to handle equal/not equal check
Branch Target Address:
- Introduce additional logic to calculate the address
- Need PC (form Fetch) & Offset (from Decode)

Memory Stage

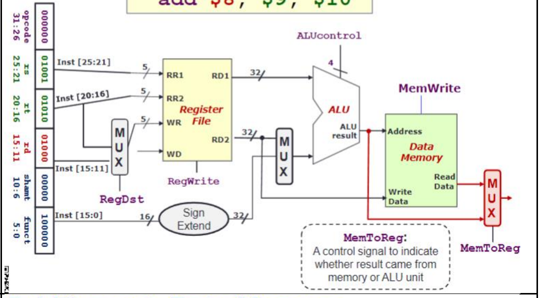
Instruction Memory Access Stage: Only for load and store instructions
- Use Memory Address calculated by ALU Stage
- Read from or write to data memory
Input from previous stage (ALU): Computation result to be used as memory address (if applicable)
Output to next stage (Register Write): Result to be stored

Memory Stage (Visualised)

5.4 Memory Stage: Non-Memory Inst.

- Add a multiplexer to choose the result to be stored

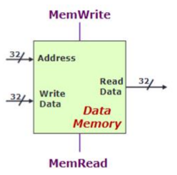
`add $8, $9, $10`



Element: Data Memory

5.4 Element: Data Memory

- Storage element for the data of a program
- **Inputs:**
  - Memory Address
  - Data to be written (Write Data) for store instructions
- **Control:**
  - Read and Write controls; only one can be asserted at any point of time
- **Output:**
  - Data read from memory (Read Data) for load instructions



As a Function

```
function DataMem(addr,WD,MW,MR) {
  if(MW) {
    Mem[addr] = WD;
  } else if(MR) {
    return Mem[addr];
  }
}
```

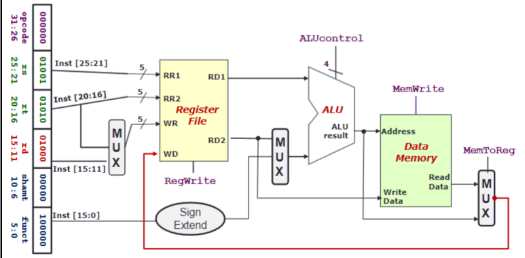
Register Write Stage

Instruction Register Write Stage:
Most instructions write the result of some computation into a register
- Arithmetic, logical, shifts, loads, set-less-than
- Need destination register number and computation result
Input from previous stage (Memory): Computation result either from memory or ALU

Register Write Stage (Visualised)

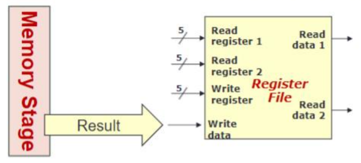
5.5 Register Write Stage: Routing

`add $8, $9, $10`



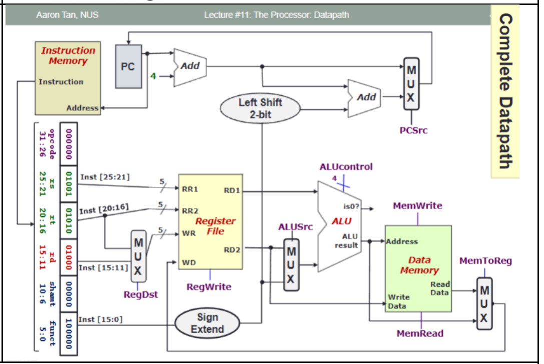
Register Write Stage: Block Diagram

5.5 Register Write Stage: Block Diagram



- Result Write stage has no additional element:
  - Basically just route the correct result into register file
  - The ***Write Register*** number is generated way back in the **Decode** Stage

Complete Datapath

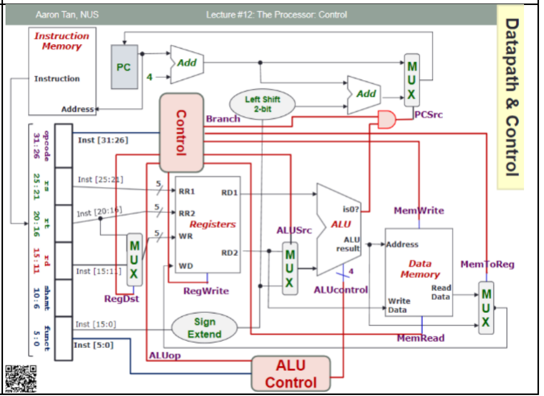


Control

Idea / Concept

Generate control signals based on instruction to be executed
Design a combinational circuit to generate control signals based on Opcode and possibly Function code
Uses a control unit

Controlpath (Visualised)



Control Signals

Control Signal	Execution Stage	Purpose
RegDst	Decode/Operand Fetch	Select the destination register number
RegWrite	Decode/Operand Fetch RegWrite	Enable writing of register
ALUSrc	ALU	Select the 2^{nd} operand for ALU
ALUcontrol	ALU	Select the operation to be performed
MemRead / MemWrite	Memory	Enable reading/writing of data memory
MemToReg	RegWrite	Select the result to be written back to register file
PCSrc	Memory/RegWrite	Select the next PC value

RegDst	False(0): Write Register = Inst[20:16] True(1): Write Register = Inst[15:11]
RegWrite	False(0): No register write True(1): New value will be written
ALUSrc	False(0): Operand2 = Register Read Data 2 True(0): Operand2 = SignExt(Inst[15:0])
MemRead	False(0): Not performing memory read access True(1): Read memory using Address
MemWrite	False(0): Not performing memory write operation True(1): memory[Address] ⇐ Register Read Data 2
MemToReg *SWAPPED	True(1): Register Write Data = Memory Read Data False(0): Register Write Data = ALU Result
PCSrc	AND gate inputs: branch inst opcode, ALU's "isZero?" False(0): Next PC = PC + 4 True(1): Next PC = SignExt(Inst[15:0]) << 2 + (PC + 4)

ALUcontrol

4-bit signal from ALU Control: operation to do in ALU

Opcode	ALUop	Instruction Operation	Funct field	ALU action	ALU control
lw	00	load word	XXXXXX	add	0010
sw	00	store word	XXXXXX	add	0010
beq	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	10 0000	add	0010
R-type	10	subtract	10 0010	subtract	0110
R-type	10	AND	10 0100	AND	0000
R-type	10	OR	10 0101	OR	0001
R-type	10	set on less than	10 1010	set on less than	0111

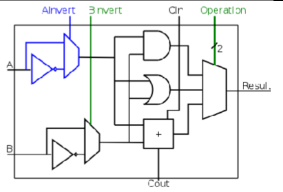
Instruction Type	ALUop
lw / sw	00
beq	01
R-type	10

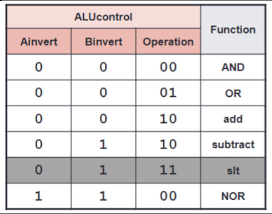
ALUcontrol	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	slt
1100	NOR

Generation of 2-bit ALUop signal will be discussed later

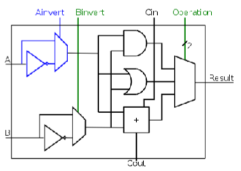
ALU (1-bit)

- 4 control bits are needed:
  - Ainvert:
    - 1 to invert input A
  - Binvert:
    - 1 to invert input B
  - Operation (2-bit)
    - To select one of the 3 results





ALUcontrol			Function
Ainvert	Binvert	Operation	
0	0	00	AND
0	0	01	OR
0	0	10	add
0	1	10	subtract
0	1	11	slt
1	1	00	NOR



Multilevel Decoding

Producing ALUcontrol signal with 6-bit Opcode & Function code
Use some of the input to reduce cases, then generate the full ouput
Simplify the design process, reduce the size of the main controller, potentially speedup the circuit

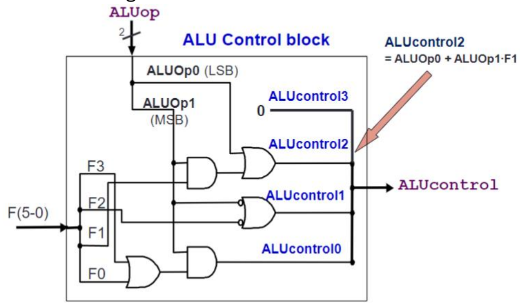
ALUop

2-bit ALUop signal
Uses Opcode to generate

Instruction Type	ALUop
lw / sw	00
beq	01
R-type	10

ALU Control

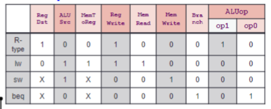
Uses ALUop signal and Function code to generate ALUcontrol signal



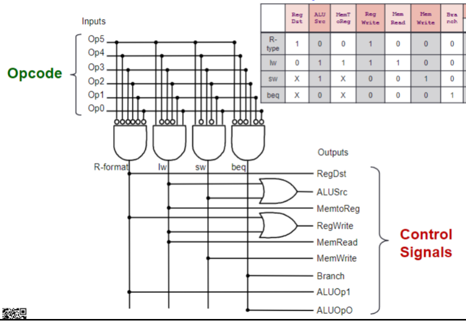
Control

Produces Control Signals based on Opcode

5. Combinational Circuit Implementation



	RegDst	ALUSrc	MemToReg	RegWrite	MemRead	MemWrite	Branch	ALUop	
								op1	op0
R-type	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1



MIPS Reference Card ("Green Card") 1. Pull along perforation to separate card 2. Fold bottom side (columns 3 and 4) together

MIPS Reference Data

①

CORE INSTRUCTION SET

NAME, MNEMONIC		FOR-MAT	OPERATION (in Verilog)		OPCODE / FUNCT (Hex)
Add	add	R	R[rd] = R[rs] + R[rt]	(1)	0 / 20_{hex}
Add Immediate	addi	I	R[rt] = R[rs] + SignExtImm	(1,2)	8_{hex}
Add Imm. Unsigned	addiu	I	R[rt] = R[rs] + SignExtImm	(2)	9_{hex}
Add Unsigned	addu	R	R[rd] = R[rs] + R[rt]		0 / 21_{hex}
And	and	R	R[rd] = R[rs] & R[rt]		0 / 24_{hex}
And Immediate	andi	I	R[rt] = R[rs] & ZeroExtImm	(3)	c_{hex}
Branch On Equal	beq	I	if(R[rs]==R[rt]) PC=PC+4+BranchAddr	(4)	4_{hex}
Branch On Not Equal	bne	I	if(R[rs]!=R[rt]) PC=PC+4+BranchAddr	(4)	5_{hex}
Jump	j	J	PC=JumpAddr	(5)	2_{hex}
Jump And Link	jal	J	R[31]=PC+8;PC=JumpAddr	(5)	3_{hex}
Jump Register	jr	R	PC=R[rs]		0 / 08_{hex}
Load Byte Unsigned	lbu	I	R[rt]={24'b0,M[R[rs] +SignExtImm](7:0)}	(2)	24_{hex}
Load Halfword Unsigned	lhu	I	R[rt]={16'b0,M[R[rs] +SignExtImm](15:0)}	(2)	25_{hex}
Load Linked	ll	I	R[rt] = M[R[rs]+SignExtImm]	(2,7)	30_{hex}
Load Upper Imm.	lui	I	R[rt] = {imm, 16'b0}		f_{hex}
Load Word	lw	I	R[rt] = M[R[rs]+SignExtImm]	(2)	23_{hex}
Nor	nor	R	R[rd] = ~ (R[rs] | R[rt])		0 / 27_{hex}
Or	or	R	R[rd] = R[rs] | R[rt]		0 / 25_{hex}
Or Immediate	ori	I	R[rt] = R[rs] | ZeroExtImm	(3)	d_{hex}
Set Less Than	slt	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0		0 / $2a_{hex}$
Set Less Than Imm.	slti	I	R[rt] = (R[rs] < SignExtImm)? 1 : 0	(2)	a_{hex}
Set Less Than Imm. Unsigned	sltiu	I	R[rt] = (R[rs] < SignExtImm) ? 1 : 0	(2,6)	b_{hex}
Set Less Than Unsig.	sltu	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0	(6)	0 / $2b_{hex}$
Shift Left Logical	sll	R	R[rd] = R[rt] << shamt		0 / 00_{hex}
Shift Right Logical	srl	R	R[rd] = R[rt] >> shamt		0 / 02_{hex}
Store Byte	sb	I	M[R[rs]+SignExtImm](7:0) = R[rt](7:0)	(2)	28_{hex}
Store Conditional	sc	I	M[R[rs]+SignExtImm] = R[rt]; R[rt] = (*atomic*) ? 1 : 0	(2,7)	38_{hex}
Store Halfword	sh	I	M[R[rs]+SignExtImm](15:0) = R[rt](15:0)	(2)	29_{hex}
Store Word	sw	I	M[R[rs]+SignExtImm] = R[rt]	(2)	$2b_{hex}$
Subtract	sub	R	R[rd] = R[rs] - R[rt]	(1)	0 / 22_{hex}
Subtract Unsigned	subu	R	R[rd] = R[rs] - R[rt]		0 / 23_{hex}

(1) May cause overflow exception
(2) SignExtImm = { 16{immediate[15]}, immediate }
(3) ZeroExtImm = { 16{1b'0}, immediate }
(4) BranchAddr = { 14{immediate[15]}, immediate, 2'b0 }
(5) JumpAddr = { PC+4[31:28], address, 2'b0 }
(6) Operands considered unsigned numbers (vs. 2's comp.)
(7) Atomic test&set pair; R[rt] = 1 if pair atomic, 0 if not atomic

BASIC INSTRUCTION FORMATS

	31–26	25–21	20–16	15–11	10–6	5–0
R	opcode	rs	rt	rd	shamt	funct
I	opcode	rs	rt	immediate		
J	opcode	address				

②

ARITHMETIC CORE INSTRUCTION SET

NAME, MNEMONIC		FOR-MAT	OPERATION		OPCODE / FMT /FT / FUNCT (Hex)
Branch On FP True	bc1t	FI	if(FPcond)PC=PC+4+BranchAddr	(4)	11/8/1/--
Branch On FP False	bc1f	FI	if(!FPcond)PC=PC+4+BranchAddr	(4)	11/8/0/--
Divide	div	R	Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]		0/--/--/1a
Divide Unsigned	divu	R	Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]	(6)	0/--/--/1b
FP Add Single	add.s	FR	F[fd]= F[fs] + F[ft]		11/10/--/0
FP Add Double	add.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} + {F[ft],F[ft+1]}		11/11/--/0
FP Compare Single	c.*x*.s*	FR	FPcond = (F[fs] *op* F[ft]) ? 1 : 0		11/10/--/*y*
FP Compare Double	c.*x*.d*	FR	FPcond = ({F[fs],F[fs+1]} *op* {F[ft],F[ft+1]}) ? 1 : 0		11/11/--/*y*
FP Divide Single	div.s	FR	F[fd] = F[fs] / F[ft]		11/10/--/3
FP Divide Double	div.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} / {F[ft],F[ft+1]}		11/11/--/3
FP Multiply Single	mul.s	FR	F[fd] = F[fs] * F[ft]		11/10/--/2
FP Multiply Double	mul.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} * {F[ft],F[ft+1]}		11/11/--/2
FP Subtract Single	sub.s	FR	F[fd]=F[fs] - F[ft]		11/10/--/1
FP Subtract Double	sub.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} - {F[ft],F[ft+1]}		11/11/--/1
Load FP Single	lwc1	I	F[rt]=M[R[rs]+SignExtImm]	(2)	31/--/--/--
Load FP Double	ldc1	I	F[rt]=M[R[rs]+SignExtImm]; F[rt+1]=M[R[rs]+SignExtImm+4]	(2)	35/--/--/--
Move From Hi	mfhi	R	R[rd] = Hi		0 /--/--/10
Move From Lo	mflo	R	R[rd] = Lo		0 /--/--/12
Move From Control	mfc0	R	R[rd] = CR[rs]		10 /0/--/0
Multiply	mult	R	{Hi,Lo} = R[rs] * R[rt]		0/--/--/18
Multiply Unsigned	multu	R	{Hi,Lo} = R[rs] * R[rt]	(6)	0/--/--/19
Shift Right Arith.	sra	R	R[rd] = R[rt] >>> shamt		0/--/--/3
Store FP Single	swc1	I	M[R[rs]+SignExtImm] = F[rt]	(2)	39/--/--/--
Store FP Double	sdc1	I	M[R[rs]+SignExtImm] = F[rt]; M[R[rs]+SignExtImm+4] = F[rt+1]	(2)	3d/--/--/--

* (*x* is eq, lt, or le) (*op* is ==, <, or <=) (*y* is 32, 3c, or 3e)

FLOATING-POINT INSTRUCTION FORMATS

	31–26	25–21	20–16	15–11	10–6	5–0
FR	opcode	fmt	ft	fs	fd	funct
FI	opcode	fmt	ft	immediate		

PSEUDOINSTRUCTION SET

NAME	MNEMONIC	OPERATION
Branch Less Than	blt	if(R[rs]<R[rt]) PC = Label
Branch Greater Than	bgt	if(R[rs]>R[rt]) PC = Label
Branch Less Than or Equal	ble	if(R[rs]<=R[rt]) PC = Label
Branch Greater Than or Equal	bge	if(R[rs]>=R[rt]) PC = Label
Load Immediate	li	R[rd] = immediate
Move	move	R[rd] = R[rs]

REGISTER NAME, NUMBER, USE, CALL CONVENTION

NAME	NUMBER	USE	PRESERVED ACROSS A CALL?
$zero	0	The Constant Value 0	N.A.
$at	1	Assembler Temporary	No
$v0-$v1	2-3	Values for Function Results and Expression Evaluation	No
$a0-$a3	4-7	Arguments	No
$t0-$t7	8-15	Temporaries	No
$s0-$s7	16-23	Saved Temporaries	Yes
$t8-$t9	24-25	Temporaries	No
$k0-$k1	26-27	Reserved for OS Kernel	No
$gp	28	Global Pointer	Yes
$sp	29	Stack Pointer	Yes
$fp	30	Frame Pointer	Yes
$ra	31	Return Address	Yes

MIPS Reference Data Card ("Green Card")

1. Pull along perforation to separate card 2. Fold bottom side (columns 3 and 4) together

OPCODES, BASE CONVERSION, ASCII SYMBOLS

(3)

MIPS opcode (31:26)	(1) MIPS funct (5:0)	(2) MIPS funct (5:0)	Binary	Deci-mal	Hexa-deci-mal	ASCII Char-acter	Deci-mal	Hexa-deci-mal	ASCII Char-acter
(1)	sll	add.*f*	00 0000	0	0	NUL	64	40	@
		sub.*f*	00 0001	1	1	SOH	65	41	A
j	srl	mul.*f*	00 0010	2	2	STX	66	42	B
jal	sra	div.*f*	00 0011	3	3	ETX	67	43	C
beq	sllv	sqrt.*f*	00 0100	4	4	EOT	68	44	D
bne		abs.*f*	00 0101	5	5	ENQ	69	45	E
blez	srlv	mov.*f*	00 0110	6	6	ACK	70	46	F
bgtz	srav	neg.*f*	00 0111	7	7	BEL	71	47	G
addi	jr		00 1000	8	8	BS	72	48	H
addiu	jalr		00 1001	9	9	HT	73	49	I
slti	movz		00 1010	10	a	LF	74	4a	J
sltiu	movn		00 1011	11	b	VT	75	4b	K
andi	syscall	round.w.*f*	00 1100	12	c	FF	76	4c	L
ori	break	trunc.w.*f*	00 1101	13	d	CR	77	4d	M
xori		ceil.w.*f*	00 1110	14	e	SO	78	4e	N
lui	sync	floor.w.*f*	00 1111	15	f	SI	79	4f	O
	mfhi		01 0000	16	10	DLE	80	50	P
(2)	mthi		01 0001	17	11	DC1	81	51	Q
	mflo	movz.*f*	01 0010	18	12	DC2	82	52	R
	mtlo	movn.*f*	01 0011	19	13	DC3	83	53	S
			01 0100	20	14	DC4	84	54	T
			01 0101	21	15	NAK	85	55	U
			01 0110	22	16	SYN	86	56	V
			01 0111	23	17	ETB	87	57	W
	mult		01 1000	24	18	CAN	88	58	X
	multu		01 1001	25	19	EM	89	59	Y
	div		01 1010	26	1a	SUB	90	5a	Z
	divu		01 1011	27	1b	ESC	91	5b	[
			01 1100	28	1c	FS	92	5c	\
			01 1101	29	1d	GS	93	5d	]
			01 1110	30	1e	RS	94	5e	^
			01 1111	31	1f	US	95	5f	_
lb	add	cvt.s.*f*	10 0000	32	20	Space	96	60	`
lh	addu	cvt.d.*f*	10 0001	33	21	!	97	61	a
lwl	sub		10 0010	34	22	"	98	62	b
lw	subu		10 0011	35	23	#	99	63	c
lbu	and	cvt.w.*f*	10 0100	36	24	$	100	64	d
lhu	or		10 0101	37	25	%	101	65	e
lwr	xor		10 0110	38	26	&	102	66	f
	nor		10 0111	39	27	'	103	67	g
sb			10 1000	40	28	(	104	68	h
sh			10 1001	41	29	)	105	69	i
swl	slt		10 1010	42	2a	*	106	6a	j
sw	sltu		10 1011	43	2b	+	107	6b	k
			10 1100	44	2c	,	108	6c	l
			10 1101	45	2d	-	109	6d	m
swr			10 1110	46	2e	.	110	6e	n
cache			10 1111	47	2f	/	111	6f	o
ll	tge	c.f.*f*	11 0000	48	30	0	112	70	p
lwc1	tgeu	c.un.*f*	11 0001	49	31	1	113	71	q
lwc2	tlt	c.eq.*f*	11 0010	50	32	2	114	72	r
pref	tltu	c.ueq.*f*	11 0011	51	33	3	115	73	s
	teq	c.olt.*f*	11 0100	52	34	4	116	74	t
ldc1		c.ult.*f*	11 0101	53	35	5	117	75	u
ldc2	tne	c.ole.*f*	11 0110	54	36	6	118	76	v
		c.ule.*f*	11 0111	55	37	7	119	77	w
sc		c.sf.*f*	11 1000	56	38	8	120	78	x
swc1		c.ngle.*f*	11 1001	57	39	9	121	79	y
swc2		c.seq.*f*	11 1010	58	3a	:	122	7a	z
		c.ngl.*f*	11 1011	59	3b	;	123	7b	{
		c.lt.*f*	11 1100	60	3c	<	124	7c	|
sdc1		c.nge.*f*	11 1101	61	3d	=	125	7d	}
sdc2		c.le.*f*	11 1110	62	3e	>	126	7e	~
		c.ngt.*f*	11 1111	63	3f	?	127	7f	DEL

(1) opcode(31:26) == 0
(2) opcode(31:26) == 17_{ten} (11_{hex}); if fmt(25:21)==16_{ten} (10_{hex}) *f* = s (single);
if fmt(25:21)==17_{ten} (11_{hex}) *f* = d (double)

IEEE 754 FLOATING-POINT STANDARD

(4)

$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$

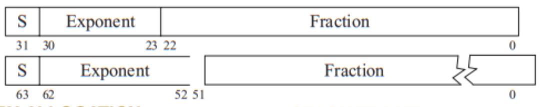
where Single Precision Bias = 127,
Double Precision Bias = 1023.

IEEE 754 Symbols

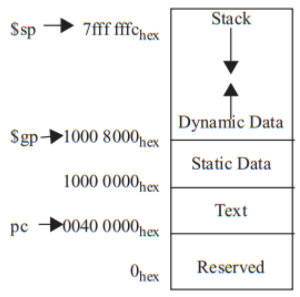
Exponent	Fraction	Object
0	0	± 0
0	≠0	± Denorm
1 to MAX - 1	anything	± Fl. Pt. Num.
MAX	0	±∞
MAX	≠0	NaN

S.P. MAX = 255, D.P. MAX = 2047

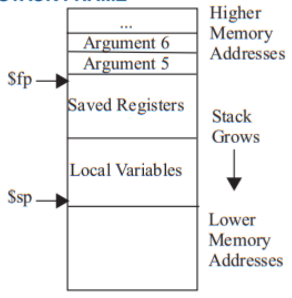
IEEE Single Precision and Double Precision Formats:



MEMORY ALLOCATION



STACK FRAME

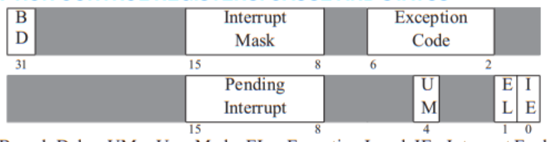


DATA ALIGNMENT

Double Word							
Word				Word			
Halfword		Halfword		Halfword		Halfword	
Byte	Byte	Byte	Byte	Byte	Byte	Byte	Byte
0	1	2	3	4	5	6	7

Value of three least significant bits of byte address (Big Endian)

EXCEPTION CONTROL REGISTERS: CAUSE AND STATUS



BD = Branch Delay, UM = User Mode, EL = Exception Level, IE =Interrupt Enable

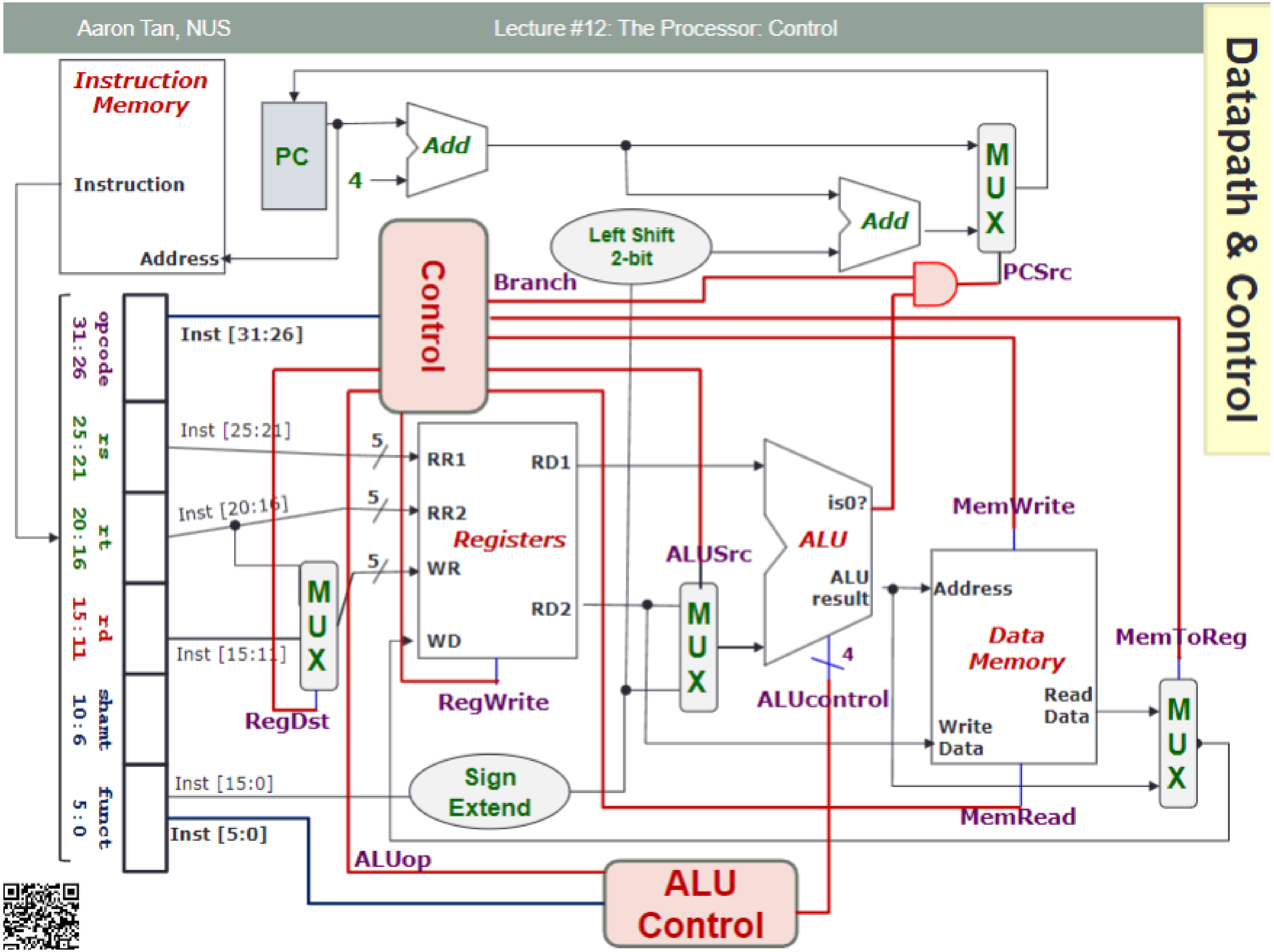
EXCEPTION CODES

Number	Name	Cause of Exception	Number	Name	Cause of Exception
0	Int	Interrupt (hardware)	9	Bp	Breakpoint Exception
4	AdEL	Address Error Exception (load or instruction fetch)	10	RI	Reserved Instruction Exception
5	AdES	Address Error Exception (store)	11	CpU	Coprocessor Unimplemented
6	IBE	Bus Error on Instruction Fetch	12	Ov	Arithmetic Overflow Exception
7	DBE	Bus Error on Load or Store	13	Tr	Trap
8	Sys	Syscall Exception	15	FPE	Floating Point Exception

SIZE PREFIXES (10^x for Disk, Communication; 2^x for Memory)

SIZE	PRE-FIX	SIZE	PRE-FIX	SIZE	PRE-FIX	SIZE	PRE-FIX
10^3, 2^{10}	Kilo-	10^{15}, 2^{50}	Peta-	10^{-3}	milli-	10^{-15}	femto-
10^6, 2^{20}	Mega-	10^{18}, 2^{60}	Exa-	10^{-6}	micro-	10^{-18}	atto-
10^9, 2^{30}	Giga-	10^{21}, 2^{70}	Zetta-	10^{-9}	nano-	10^{-21}	zepto-
10^{12}, 2^{40}	Tera-	10^{24}, 2^{80}	Yotta-	10^{-12}	pico-	10^{-24}	yocto-

The symbol for each prefix is just its first letter, except μ is used for micro.


Aaron Tan, NUS
Lecture #12: The Processor: Control
Datapath & Control
Instruction Memory
Instruction
Address
PC
Add
4
Add
MUX
PCSrc
Left Shift 2-bit
Control
Branch
Inst [31:26]
Inst [25:21]
Inst [20:16]
Inst [15:11]
Inst [15:0]
Inst [5:0]
opcode 31:26
rs 25:21
rt 20:16
rd 15:11
shamt 10:6
funct 5:0
5
5
5
RR1
RR2
WR
WD
RD1
RD2
Registers
MUX
RegDst
RegWrite
ALUSrc
MUX
is0?
ALU
ALU result
4
ALUcontrol
MemWrite
Address
Data Memory
Read Data
Write Data
MemRead
MemToReg
MUX
Sign Extend
ALUop
ALU Control