CS2100 CheatSheet

by Zachary Chua

C programming

Variables

Contains name, data type, address and value (modifiable)
Declaration and initialisation: `int count = 3`
Without initialisation, vairable contains unknown value, not 0

Data Types

1. int (integer): 4 bytes, -2,147,483,648 (-2^{31}) to 2,147,483,647 ($2^{31}-1$)
2. float (real number): 4 bytes
3. double (real number): 8 bytes
4. char (character): 1 byte, enclosed in pair of single quotes eg. 'a'

Preprocessor Directives

1. Include libraries eg. `#include <math.h>`, need to compile with `-lm` if imported math lib
2. Macro expansions eg. `#define PI 3.142 //use CAPS for macro` Macro expansions do a textual substitution

Input/Output

Input/Output statements:
- scanf(format string, input list);
- printf(format string);
- printf(format string, print list)

Format Specifiers

Placeholder	Variable Type	Function Use
%c	char	printf / scanf
%d	int	printf / scanf
%f	float or double	printf
%f	float	printf
%lf	double	scanf
%e	float or double	printf (for scientific notation)

Examples
- %5d: displays integer with width 5, right justified
- %8.3f: display real number, width of 8, 3dp, right justified

Operators

Equals operator

= has the side effect of returning the value assigned

Arithmetic Operators and Precedence

Operator Type	Operator	Associativity
Primary expression operators	`( )` *expr++ expr--*	Left to right
Unary operators	`* & + -` *++expr --expr (typecast)*	Right to left
Binary operators	`* / %`	Left to right
	`+ -`	
Assignment operators	`= += -= *= /= %=`	Right to left

Mixed-Type Arithmetic Operations
- `int m = 10/4;` means `m = 2;`
- `float p = 10/4;` means `p = 2.0;`
- `int n = 10/4.0` means `n = 2;`
- `float q = 10/4.0` means `q = 2.5;`
- `r = -10/4.0;` means `r = -2;`

Type Casting

syntax: `(type) expression`

```
int aa = 6; float ff = 15.8;
float pp = (float) aa / 4;     means  pp = 1.5;
int   nn = (int) ff / aa;      means  nn = 2;
float qq = (float) (aa / 4);   means  qq = 1.0;
```

Remainder

% is remainder in C
- `a = 10 % 4` → `a = 2`
- `a = -10 % 4` → `a = -2`

Booleans

No boolean types in C, use integers ie
- 0 represents false
- any other integer (usually 1) represents true

Operator Precedence

Operator Type	Operator	Associativity
Primary expression operators	`( ) [ ] . -> expr++ expr--`	Left to Right
Unary operators	`* & + - ! ~ ++expr --expr (typecast) sizeof`	Right to Left
Binary operators	`* / %`	Left to Right
	`+ -`	
	`< > <= >=`	
	`== !=`	
	`&&`	
	`||`	
Ternary operator	`?:`	Right to Left
Assignment operators	`= += -= *= /= %=`	Right to Left

Python: `cond ? expr1 : expr2` → `expr1 if cond else cond2`

Note that precedence of && is greater than that of ||
&& and || use short-circuit evaluation

Loops

```
while ( condition )
{
   // loop body
}
```

```
do
{
   // loop body
} while ( condition );
```

```
for ( initialization; condition; update )
{
    // loop body
}
```

Initialization: initialize the **loop variable**

Condition: repeat loop while the condition on **loop variable** is true

Update: change value of **loop variable**

For `for` loops in C, declaration of the loop variable has to be before the `for` loop

Number Systems

Data Representation

- bit: 0 or 1
- byte: 8 bits
- word: Multiple of bytes, 4 for mips32

N bits can represent up to 2^N values, from $0-(2^N-1)$
To represent M values, $\lceil log_2^M \rceil$ bits are required
In C,
- Prefix 0 for octal, eg 032 represents $(32)_8$
- Prefix 0x for hexadecimal, eg $0x32$ represents $(32)_16$

Convertion from Decimal to Base R

- Whole numbers: repeated division by R until quotient 0

eg for decimal to binary

$(43)_{10} = (\ 101011\ )_2$

2 | 43		
2 | 21	rem 1	← LSB
2 | 10	rem 1	
2 | 5	rem 0	
2 | 2	rem 1	
2 | 1	rem 0	
0	rem 1	← MSB

- Fractions: repeated multiplication by R until fractional product is 0, taking the carry

eg decimal to binary

$(0.3125)_{10} = (\ .0101\ )_2$

	Carry	
$0.3125\times2=0.625$	0	←MSB
$0.625\times2=1.25$	1	
$0.25\times2=0.50$	0	
$0.5\times2=1.00$	1	←LSB

Binary to Octal and Hexa shortcuts

- Binary to Octal: partition in groups of 3, take the value of the groups
- Octal to Binary: reverse - Binary to Hexa: partition in groups of 4, take the value of the groups
- Hexa to Binary: reverse, convert hexa to binary, extend to 4 digits if needed

ASCII code

Integers (0 – 127) and characters are 'somewhat' interchangeable in C

Negative Numbers

3 representations
1. Sign-and-Magnitude
2. 1s complement
3. 2s complement

Sign and Magnitude

Sign is represented by sign bit; 0 for + and 1 for −
First bit is sign bit, other 7 bits are read as normal
- Largest value: $2^{n-1}-1$
- Smallest value: $-2^{n-1}+1$

- 2 zeroes: $+0 = 00000000$ and $-0 = 10000000$
- Range: $2^{n-1} - 1$ to $-2^{n-1} + 1$

To negate the number, just invert sign bit

1s Complement

Negated value of x is given by $-x = 2^n - x - 1$
- Largest value: $2^{n-1} - 1$
- Smallest value" $-2^{n-1} - 1$
- 2 Zeroes: $+0 = 00000000$ and $-0 = 11111111$
- Range: $-(2^{n-1} - 1)$ to $2^{n-1} - 1$

To negate a number, invert all bits

Note that the first bit still represents the sign: 0 for positive and 1 for negative.

2s Complement

Negated value of x is given by $-x = 2^n - x$
- Largest value: $2^{n-1} - 1$
- Smallest value: -2^{n-1}
- Range: -2^{n-1} to $2^{n-1} - 1$

To negate a number, invert all bits, then add 1

For fractions same thing, flip then add one to LSB

Note that MSB still represents the sign, and that MSB "has a value" of -2^{n-1}

Sign Extension

For 1s and 2s complement: Extend sign bit

For SaM: Pad 0s after the sign bit

Arithmetics

2s Complement Addition and Subtraction

Algorithm for adding integers A and B

1. Perform binary addition on the two numbers
2. Ignore the carry out of MSB
3. Check for overflow, overflow occurs if the 'carry in' and 'carry out' of the MSB are different. Or if the result is opposite sign of A and B

Algorithm for Subtraction of A and B, $A - B = A + (-B)$

1. Take 2s complement of B
2. Add 2s complement of B to A, using above algorithm

1s Complement Addition and Subtraction

Algorithm for adding integers A and B

1. Perform binary addition on the two numbers
2. If there is a carry out of the MSB, add 1 to the result (at the LSB)
3. Check for overflow

Algorithm for subtraction of A and B

1. Take 1s complement of B
2. Add the 1s complement of B to A

Overflows

Overflows are a result of addition/subtraction going beyong the range of numbers
- positive add positive → negative
- negative add negative → positive

Excess Representation

Allows the range of values to be distributed evenly between positive and negative values, using a simple translation (subtraction)

$$\text{Excess representation} = \text{Value} + \text{excess}$$

eg for excess-8, rep = Value + 8

For 4-bit numbers, usually use excess-7 or excess-8

Real numbers

Numbers with fractional components

Fixed point representation

Number of bits allocated for whole number part and fractional part are fixed
- Advantage: Easier computation
- Disadvantage: smaller range for a given precision

Floating point representation

IEEE 754 Floating-Point Representation

3 components: sign, exponent, mantissa (fraction)

Similar to standard form

Radix is assumed to be 2

1. Single Precision (32-bit): 1-bit sign, 8-bit exponent with excess-127, 23-bit mantissa
2. Double Precision (64-bit): 1-bit sign, 11-bit exponent with excess-1023, 52-bit mantissa

Mantissa normalised with an implicit leading bit 1

eg. -6.5 in decimal

$$-6.5_{10} = -110.1_2 = -1.101_2 * 2^2$$

IEEE 754: 1 10000001 10100000000000000000000$_2$ = C0D00000$_{16}$

Pointers and Functions

Pointers

Variables occupies some space in memory and has an address

Can refer to the address of a variable by usiing the *address of operator*, &

%p is used as the format specifiier for addressses, addresses are printed out in hexadecimal format

Pointer Variables

Variable that contains the address of another variable.

If pointer variable, a_ptr, contains address of variable a

a_ptr is said to be pointing to a

Declaring a Pointer

Syntax:

```
type *pointer_name
```

- pointer_name is the name of the pointer (Good practice to name a pointer with suffix _p or _ptr)
- type is the data type of the variable this pointer may point to

eg. `int *a_ptr` declares a pointer to an int named `a_ptr`

Assigning Value to a Pointer

```
int a = 123;
int *a_ptr; // declaring an int pointer

a_ptr = &a;
```

```
int a = 123;
int *a_ptr = &a; // initialising a_ptr
```

Accessing Variable through Pointer

Once a_ptr points to a, can access a through a_ptr using *indirection operator*, (or dereferencing operator) *

ie. a_ptr* is synonymous with a

Note that
- `int *a_ptr` is a declaration of a pointer to an int with the name a_ptr
- `*a_ptr` is the value AT the address stored in a_ptr

Incrementing Pointers

Recall that
- int takes up 4 bytes
- float takes up 4 bytes
- char takes up 1 byte
- double takes up 8 bytes

Incrementing a pointer means that the pointer will look at the NEXT chunk of data,

ie. incrementing an int/ float ptr increases value by 4, incrementing a char ptr increases value by 1, incrementing double ptr increases value by 8

Note remember to assign the pointer variable an address before using if not it would be pointing somewhere unknown

User Defined Functions

Function Definitions follow the following syntax

```
return_type name(p1_type p1_name, p2_type p2_name, ...) {
    // function body
}
```

Function prototypes follow the following syntax (names of params not needed)

```
return_type name(p1_type, p2_type, ...);
```

Good practice to put function prototypes at the topm before main() function, after preprocessor directives. Function definitions after main() function

Without function prototype, compiler assumes default return type of int

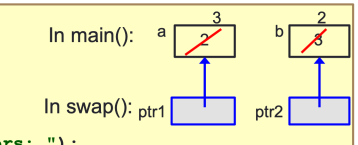
Pass-by-Value and Scope Rule

In C, actual parameters are passed to formal parameters by a mechanism called *pass-by-value*
- Formal parameters and variables are local to the function they are declared in
- Local parameters and variables are only accessible in the function they are

declared in (Scope rule)
- When function is called, activation record is created in call stack and memory is allocated for the local paramters and variables of the function
- Once done, activation record is removed, memory allocated is released
- Local params and variables of a function exist in memory only during execution of function and are known as *automatic variables*
- In contrast, *static variables* live in memory even after the function is executed

Then how to allow a function to return more than one value or modify values of variables defined outside of it?
- Using functions with pointer parameters
- eg. Swap function

```
#include <stdio.h>
void swap(int *, int *);
int main(void) {
   int a, b;
   printf("Enter two integers: ");
   scanf("%d %d", &var1, &var2);
   swap( &a, &b );
   printf("var1 = %d; var2 = %d\n", var1, var2);
   return 0;
}
void swap(int *ptr1, int *ptr2) {
   int temp;
   temp = *ptr1; *ptr1 = *ptr2; *ptr2 = temp;
}
```



SwapCorrect.c

Arrays. Strings, Structures

Arrays

Declaration: element type, array name size, eg, `int c[30];`
Initializing: arrays can be initialized at time of declaration

```
// a[0]=54, a[1]=9, a[2]=10
int a[3] = {54, 9, 10};

// size of b is 3 with b[0]=1, b[1]=2, b[2]=3
int b[] = {1, 2, 3};

// c[0]=17, c[1]=3, c[2]=10, c[3]=0, c[4]=0
int c[5] = {17, 3, 10};
```

Note what happens when fewer initial values are provided.

```
int e[2] = {1, 2, 3};  // warning issued: excess elements

int f[5];
f[5] = {8, 23, 12, -3, 6}; // too late to do this;
                           // compilation error
```

array name refers to the address of the first element
eg. `int a[10]; // a = &a[0]`

Array Assignment

Array name is a fixed pointer, it points to the first element in the array and cannot be altered

```
int source[10] = { 10, 20, 30, 40, 50 };
int dest[10];
dest = source; //illegal
```

Instead use a loop to loop through elements and copy them over, or use `memcpy()` in <string.h> library

Array params in functions

Function prototype: `int sumArray(int [], int);`
Function Definition:

```
int sumArray(int arr[], int size) { ... }
int sumArray(int arr[8], int size) { ... }
```

- Number 8 is ignored by the compiler because array parameters are passed in as pointers, must pass in size as a separate parameter
- Therefore, alternative function prototype: `int sumarray(int *, size)` and alternative function definition: `int sumArray(int *arr, int size) {...}`
- This also means that any function can modify an array it receives

Strings

- Array of chars ≠ string, need to append null character
- String is an array of chars terminated by a null character, (\0, ascii value 0)

```
char fruit_name[] = "apple";
char fruit_name[] = {'a','p','p','l','e','\0'};
```

String IO

Input:
1. `fgets(str, size, stdin) // reads size - 1 char, or until (including) newline`
2. `scanf("%s", str); // reads until whitespace`
- Don't use `gets()`, can result in buffer overflow
- `fgets()` can reads in newline character if there is enough space, need to replace it with null character

```
fgets(str, size, stdin);
len = strlen(str);
if (str[len - 1] == '\n') {
    str[len - 1] = '\0';
}
```

Output:
1. `puts(str); // terminates with new line`
2. `printf("%s\n", str);`

String Functions

1. `strlen(s)`, returns number of chars in s
2. `strcmp(s1, s2)`, compares ASCII values of chars in s1 and s2, return negative int if s1 < s2 and positive if s1 > s2, 0 if same
3. `strncmp(s1, s2, n)`, compares first n characters of s1 and s2
4. `strcpy(s1, s2)`, copies s2 into s1, must use this because cannot assign, unless at declaration. If s2 too long can cause buffer overflow
5. `strncpy(s1, s2, n)`, copies first n chars from s2 to s1

All the functions use null character, without it could result in illegal access of memory

Structures

Structures allow grouping of heterogeneous members (of different types)
A group can be a member of another group
Groups are also called structure types

Struct Definition:

```
typedef struct {
    int acctNum;
    float balance;
} account_t;
```

- Remember the ; at the end!
- A type is NOT a variable
- Types must be defined before declaring variables of that type
- No memory is allocated to a type
- Put struct definitions before function prototypes but after preprocessor directives

Accessing members of astructure variable

- Use the dot (.) operator
eg:

```
result_t result;
result.stuNum = 123456;
result.score = 62.0;
result.grade = 'D';

card_t card = { 123456, {30, 6} };
card.expiryDate.year = 2021;
```

Reading a Structure member

Structure members are read in individually

```
result_t result;
printf("Enter student number, score and grade: ");
scanf("%d %f %c", &result.stuNum, &result.score, &result.grade);
```

Unlike arrays, can do assignment with structures, eg if result has already been initialised, result.stuNum = result1.stuNum is valid

Stuctures and Functions

Returning structure from function
Can return a struct as you would for other types

Structures as parameters
- Uses pass-by-value, entire structure is copied over
- To modify a structure's content, need to pass address of structure to function, using & operator
- Therefore to access members need to use (*player).name, for example. - Note that dot operator (.) has higher precedence than indirection operator (*), therefore the () are needed
- (*ptr_name).member can be rewritten as ptr_name–>member

MIPS

Overview

Mips uses Load-Store architecture, ie. both instructions and data are stored in memory
- Limit memory operations
- Rely on registers (mips has 32) for storage during execution
Note that Registers have no data type!!
Most of mips operations are register to register

Arithmetic Operations
Addition and Subtraction
eg. textttadd $s0, $s1, $s2, adds $s1 and $s2 and stores in $s0
eg. textttsub $s0, $s1, $s2, subtracts $s2 from $s1 and stores in $s0

Immediate operations
Immediate values are constants, 16-bit, source2 is a constant instead
Values ranges from $[-2^{15}$ to $2^{15}-1]$, 16-bit 2s comp

Register Zero
$0 always has a value of 0
Can be used to assign values eg `f = g`, by writing `add $s0, $s1, $zero`, f in $s0, g in $s1

Logical Operators
View Registers as 32 raw bits instead of a number

Logical operation	C operator	Java operator	MIPS instruction
Shift Left	<<	<<	`sll`
Shift right	>>	>>, >>>	`srl`
Bitwise AND	&	&	`and, andi`
Bitwise OR	|	|	`or, ori`
Bitwise NOT	~	~	`nor`

AND

a	b	a AND b
0	0	0
0	1	0
1	0	0
1	1	1

OR

a	b	a OR b
0	0	0
0	1	1
1	0	1
1	1	1

NOR

a	b	a NOR b
0	0	1
0	1	0
1	0	0
1	1	0

XOR

a	b	a XOR b
0	0	0
0	1	1
1	0	1
1	1	0

Shift Left
- sll (shift left logical): Move all bits in a word to the left by a number of positions; fill the emptied positions with zeroes.
- has the effect of multiplying number by 2^n, where n is the number in the immediate field
- eg `sll $s1 $s1`, 3 $\rightarrow$ multiply value in s1 by 8

Shift Right
- srl (shift right logical): Move all bits in a word to the right by a number of positions; fills the emptied positions with zeroes.
- divide number by 2^n

AND
- $a \wedge 0 \implies 0$
- $a \wedge 1 \implies a$
- use above two properties for masking, set positions to be ignored to 0, set positions of interest to 1

OR
- $a \vee 0 \implies a$
- $a \vee 1 \implies 1$
- use above two properties for setting buts, set position to be set to 1

NOR
- Can be used to implement NOT instruction
- NOR with 0

XOR
- Can also be used to implement NOT instruction
- XOR with 1
- Has XORI instruction but no NORI, to keep instruction set small

Large constant
- How to load a 32 bit constant into a register? Recall instructions only have 16 bit immediate field
- use `lui` instruction to set upper 16 bits (lower 16 bits would be set to 0)
- then use `ori` instruction to set lower bits, (16 bit immediate would be 0-extended to 32-bit)
Note that logical operators 0 extend immediates to 32 bits whereas arithmetic operations sign-extend(extend first bit) CHECK THIS

Memory Organisation
- Main memory can be viewed as a large, single-dimension array of locations.
- Each location of the memory has an address (index to array), k bit address $\implies 2^k$ locations
- Each location/ address contains 1 byte (byte addressable), recall each word is 4 bytes long

Word alignment
- Words are aligned in memory if they begin at a byte address that is a multiple of the number of bytes in a word
- Consecutive words differ by **4 bytes**

Memory Instruction
Load Word
- eg `lw $t0, 4($s0)` - address = value of `$s0` + 4, eg 8000 stored in `$s0`, then address = 8004
- Memory word at `Mem[8004]` loaded into `$t0`, ie `Mem[8004] - Mem[8007]`
Store Word
- eg `sw $t0, 12($s0)` - address = value of `$s0` + 12, eg 8000 stored in `$s0`, then address = 8012
- Content of `$t0` stored into the word at `Mem[8012], ie, 8012 - 8015`

- Note that offset for both `lw` and `sw` must be a multiple of 4 (because word is 4 bytes in mips)
- MIPS does not allow unaligned load and stores with lw and sw, need to use pseudo-instructions ulw and usw
- Other load-store instructions include load byte `(lb)` and store byte `(sb)`, used for chars and char arrays, for these instructions, offset no need multiple of 4

Remember that Registers do not have type!
- `lw $t0, 0($s1)` then `$s1` should contain an address
- `add $s1 $s1 $t0` then `$s1` should containt a number value

Decisions
Like if-else statements in higher-level languages
Two types of decision-making statements in MIPS
1. Conditional (branching)
2. Unconditional (jump)

Branching
BEQ, branch on equal
- eg `beq $r1, $r2, label`
- go to the statement labeled `label` if value in $r1 is equal to value in $r2
BNE, branch not equal
- eg `bne $r1, $r2, label`
- go to the statement labeled `label` if values aren't equal

Jump
- processor always follows the branch
- eg `j label`

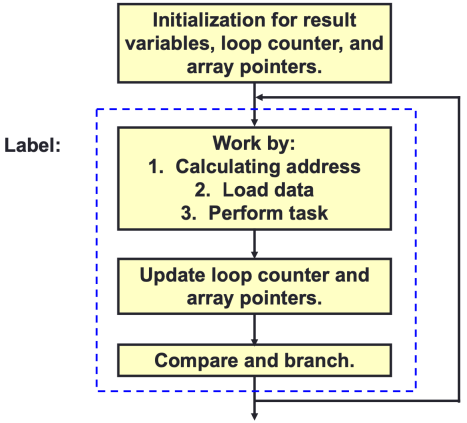
Note that label is an "anchor" in code to indicate point of interest, usually as a branch target, Labels are not instructions
- Later on labels will be converted to numbers, ie PC relative addressing for branch and truncated adress for jump
Common technique: when there is only one branch, ie if with no else, then invert the conditional for more succint code

Inequalities
For branch on less than or branch on greater than, use slt
- slt (slti): set on less than eg `slt $t0, $s1, $s2`, sets $t0 to 1 if $s1 < $s2 or 0 otherwise
- use slt with beq or bne to achieve blt or bgt (reverse $s1 and $s2 in slt)

```
blt $s1, $s2, L

slt $t0, $s1, $s2 # these two instructions equivalent to above
bne $t0, $zero, L
```

Arrays and loops



Can use pointers instead of index to make code shorter.
- index need to increment, shift left twice (if 4 bytes like int) unless array of

chars, then add to base address
- using pointers, set pointer to base address, increment by 4 each time (if int/ float etc), or increment by 1 if char array

Encoding Instructions
Constraints are to encode all instructions in 32 bits, and make them as regular as possible
1. R-format (Register format: `op $r1, $r2, $r3`)
2. I-format (Immediate format: `op $r1, $r2, Immd`)
3. J-format (Jump format: `op Immd`)

R-format
1. Opcode: 6 bits, 0 for all R-format instructions
2. rs (source register): 5 bits
3. rt (target register, second source register for R-type): 5 bits
4. rd (destination register): 5 bits
5. shamt (shift amount): 5-bits
6. funct: 6-bits

I-format
1. opcode (specifies instruction): 6-bits
2. rs (source register): 5-bits
3. rt (target register): 5-bits
4. immd: 16 bits (16 bits 2s comp signed integer)

Instruction Address
- Instructions stored in memory
- Instrcutions are 32-bits long
- Instructionsare word-aligned
Program Counter (PC) is a special register that keeps the address of the instruction being executed by processor

Branch Instruction Encoding
Immediate is only 16-bits, addresses are 32 bits. Immediate is not long enough to specify target address
But branches usually only jump small amounts, ie changes PC by small amount
- Specify target address **relative to PC**
- Since instructions are word aligned, can treat the immediate as number of *words* away from PC
- Can branch up to 2^{15} words away from PC, 2^{17} bytes
Branch Calculation:

If the branch is **not taken**:
PC = PC + 4
(**PC + 4** is address of next instruction)

If the branch is **taken**:
PC = (PC + 4) + (`immediate` × 4)

Note: immd field specifies number of words to jumps, which is the **same** as the number of instructions to skip over
Note: add immd to PC + 4 not PC due to hardware design

J-format
For jumps, can jump anywhere in memory, not small jumps so cannot use PC-relative addressing
Want to specify a 32-bit address to jump to, but cannot due to 6-bit opcode
Getting the address:
1. Start with 26-bits
2. Instructions are word aligned, so last two bits are always 00, can omit (like branching), now have 28-bit
3. Choose **4 most significant bits from PC+4**, cannot jump to anywhere in memory but should be sufficient. Max jump range: 256MB boundary

Summary
- Branches and load/store are both I-format instructions, but branch use **PC-relative addressing** while load/store uses **base addressing**
- Jumps use **pseudo-direct addressing**
- shifts use R-format, but other immediate instructions use I-format

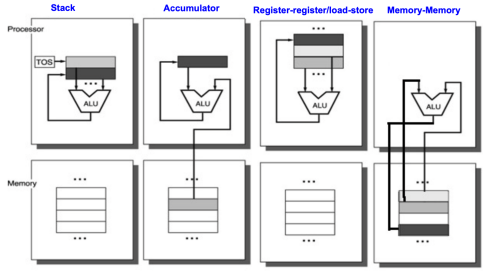
Instruction Set Architecture
5 concepts of ISA design
1. Data Storage
2. Memory Addressing Modes
3. Operations in Instruction Set
4. Instruction Formats
5. Encoding the Instruction Set

1. Data Storage
Concerns are
- Where to store the operands to that the computation can be performed?
- Where to store the result?
- How to specify the operands?

Common Storage Architectures
- Stack: Operands are implicitly on top of the stack
- Accumulator: One operand implicitly in the accumulator (special register)
- General Purpose Architecture: Only explicit operands. Register-Memory (one operand in memory) and Register-Register (load-store like MIPS)
- Memory-Memory: all operands in memory

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



2. Memory Addressing Mode
- Given k bit address, address space is of size 2^k
- Each memory transfer consists of one word of n bits
Procesor contains:
- Memory Address Register (MAR): k-bit address bus between processor and memory (one direction cpu → mem)
- Memory Data Register (MDR): n-bit data bus (bidirectional for read and write)
- control lines: eg read/write control

Endianness
The relative ordering of the bytes in a multiple-byte word stored in memory
- Big-endian: MSB stored in lowest address
- Little-endian: LSB stored in lowest address
Eg. For `0xDE AD BE EF`
- Big-endian: `0: DE, 1: AD, 2: BE, 3: EF`
- Little-endiand: `0: EF, 1: BE, 2: AD, 3: DE`
NOTE: Ordering **within** bytes are not affected, only ordering **between** bytes

Addressing Modes
In MIPS, 3 addressing modes
1. Register: Operands in register
2. Immediate: Operand is specified in instruction directly
3. Displacement: Operand is in memory with address calculated as Base + Offset (lw/sw)

4. Instruction Format
Instruction Length:
- Variable length: More flexible (but complex) and compact instruction set, requires multi-step fetch and decode
- Fixed length: Easier fetch and decode, simplify pipelining and parallelism, but instruction bits are scarce

5. Encoding the Instruction Set
Things to be decided:
- No. of registers
- No. of addressing modes
- No. of operands in instruction
Expanding Opcode scheme for fixed length instructions
- Opcode has variable length for different instructions
Eg. 16 bit fixed length instruction, 2 types of instructions, 1 operand and 2 operand, each operand takes 5 bits
- Type A: 6-bit opcode, 2∗5-bit operand
- Type B: 11-bit opcode, 1∗5-bit operand
To maximise no. of instructions: Give more 6-bit prefixes to type B
To minimise no. of instructions: Give more opcodes to type A

Datapath
Components that process data and performs arithmetic, logical and memory operations

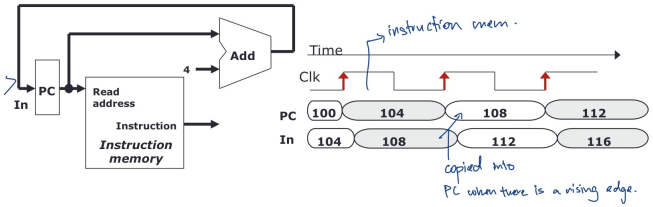
Instruction Execution Cycle
1. Instruction Fetch: Get instruction from mem (address is in PC)
2. Instruction Decode and Fetch Operands: Find out operation, get operands
3. ALU: Arithmetic and Logical operations
4. Memory Access: Memory operation (load/store)
5. Result Write: Write back result of operation to register if needed
NOTE: can merge instruction decode and operand fetch because decode is simple for MIPS

1. Instruction Fetch

1. Use PC to fetch instruction from memory
2. Increment PC by 4 to get address of next instruction (recall instructions are 32-bits long)

Exception is when branch or jump

- Sequential circuit (circuit that can store info?)
- PC is read during first half of the clock period and is updated with PC+4 ad the **next rising clock edge**, this prevents reading and updating PC at the same time

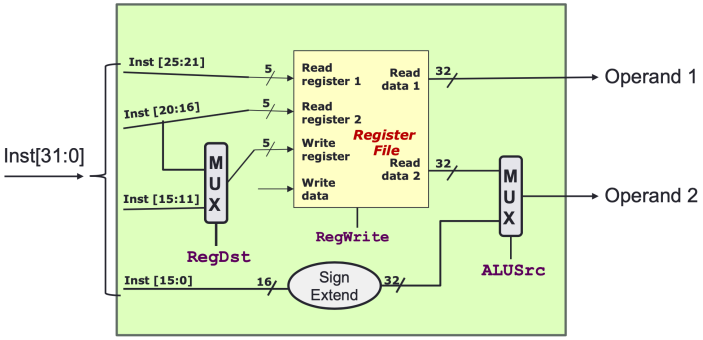


2. Instruction Decode

1. Read opcode to determine instruction type and field lengths
2. read data from necessary registers

- Register File: collection of 32 32-bit registers that can be read or written to
- Each instruction reads at most 2 registers and writes to at most 1 register
- `RegWrite` is the control signal to indicate writing fo register; 1 = write, 0 = no write
- Need to use multiplexer with `RegDst` control signal to choose between `rt` and `immediate` for write register
- Need to sign extend `immediate` to 32 bit and output to ALU

Refer to slides



3. ALU Stage aka EX stage

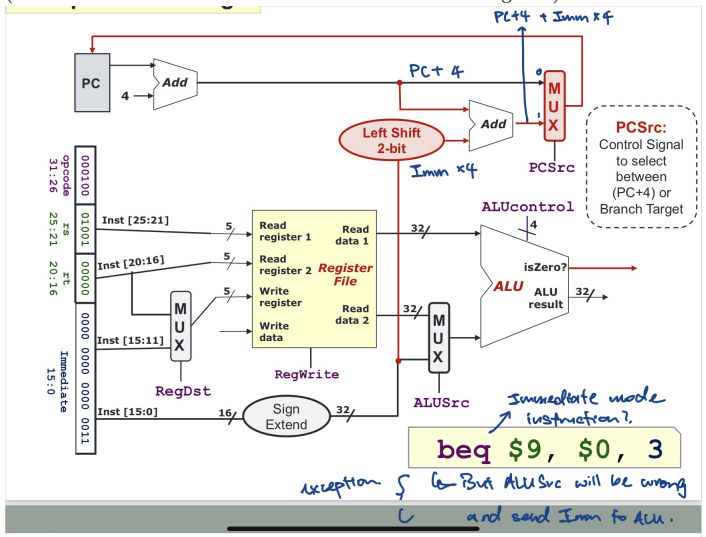
- Performs arithmetic, logical and shifting operations
- Performs register comparison and target address calculation for branching and memory operations

ALU

- Combinational logic to implement operations
- takes in 2 32 bit numbers
- outputs 32-bit result and 1 bit signal `isZero` for branching
- 4-bit control signal called `ALUcontrol`

Exception Branch instructions

- need to perform 2 calculations: Branch outcome and target address
- current circuit would be wrong because branching is I-format instruction (ALUSrc would send immediate to ALU instead of register)



4. Memory Stage

Only load and store instructions need to perform operations in this stage.

- Use memory address calculated by ALU stage
- Read or write to data memory

All other instructions remain idle

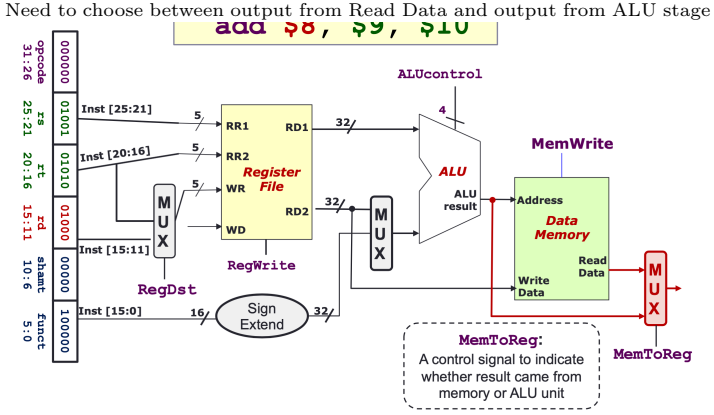
Data memory

- Inputs: memory address, data to be written (Write Data) for store instructions
- Output: Data read from memory for load instructions
- Control: Read and write controls

```
R W
1 1 undefined (should not have this case)
0 1 write
1 0 read
0 0 nothing
```

For store instructions, need to connect RD2 (rt) to Write Data.

Need to choose between output from Read Data and output from ALU stage



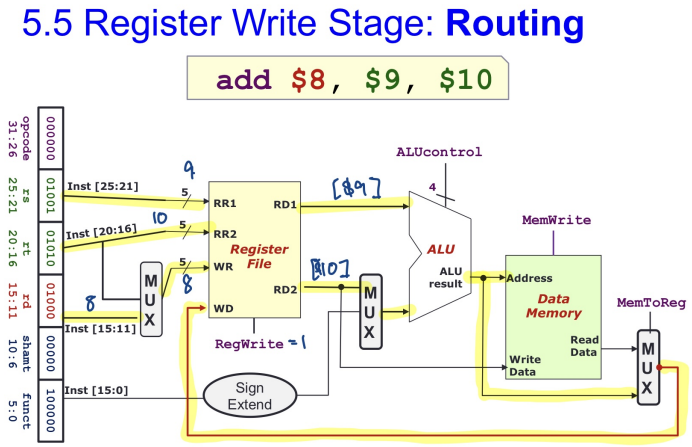
NOTE: MemToReg mux is mounted upside down ie. 1 is Read Data, 0 is ALU result

5. Register Write Stage

Write to register (stores, arithmetics, loads, slt, etc)

Just connect correct result to Register File Write Data input.

- Control: RegWrite, 1 write 0 nothing



Control

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 2nd 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

- Generate all of these signals using opcode (and funct, for R-type) using a combinational circuit

PCSrc

Note only need opcode (that it is a branch instruction) but also if the branch is taken (`isZero === 1`).

- Therefore, need to combine the two signals using `AND` gate

ALUControl

All control signals other than ALUControl can be generated from opcode only (the signals are the same for all R-type), only ALUControl needs funct code (as opcode is 0 for all R-type)

- only exception is shifts?