CS2100 CheatSheet

by Zachary Chua

C programming

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 mips

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

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

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

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

	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

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

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

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
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.

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* === 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
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, ...);
```

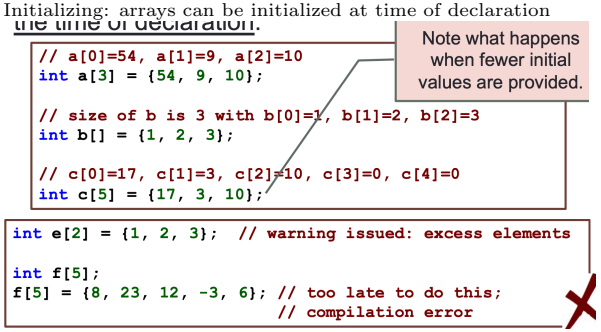
Good practice to put function prototypes at the top before main() function, after preprocessor directives. Function definitions after main() function

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 parameters 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

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



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) { ... }
```

- 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 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!
- 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 a structure 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!!

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

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
- 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)

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 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

Decisions

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. 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

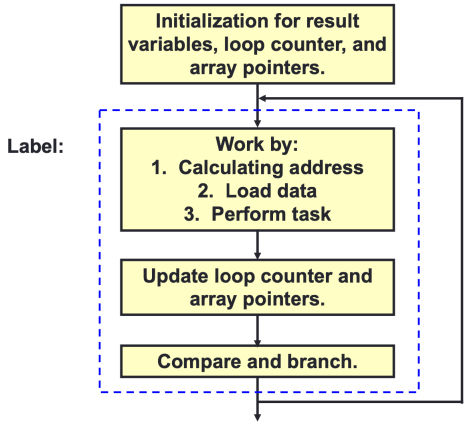
Inequalities

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`)

Instruction Address
- Instructions stored in memory
- Instructions are 32-bits long
- Instructions are 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
- 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

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

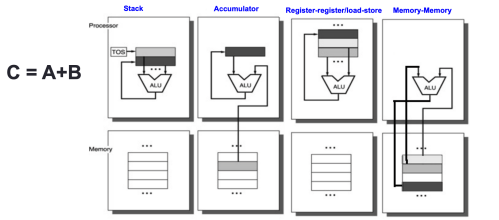
Summary
- Branches and load/store are both I-format instructions, but branch use **PC-relative addressing** while load/store uses **base addressing**
- Jumps use **pseudo-direct addressing**
- shifts use R-format, but other immediate instructions use I-format

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

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	



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

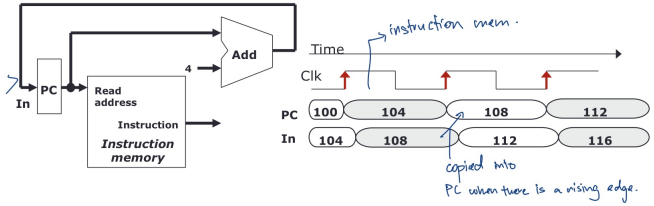
Datapath

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

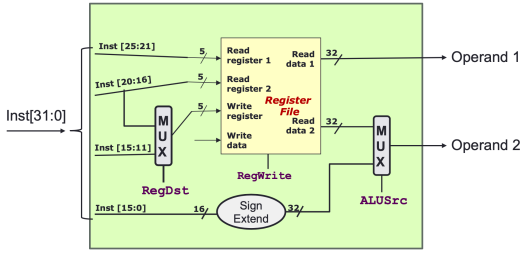
NOTE: can merge instruction decode and operand fetch because decode is simple for MIPS

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
- 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

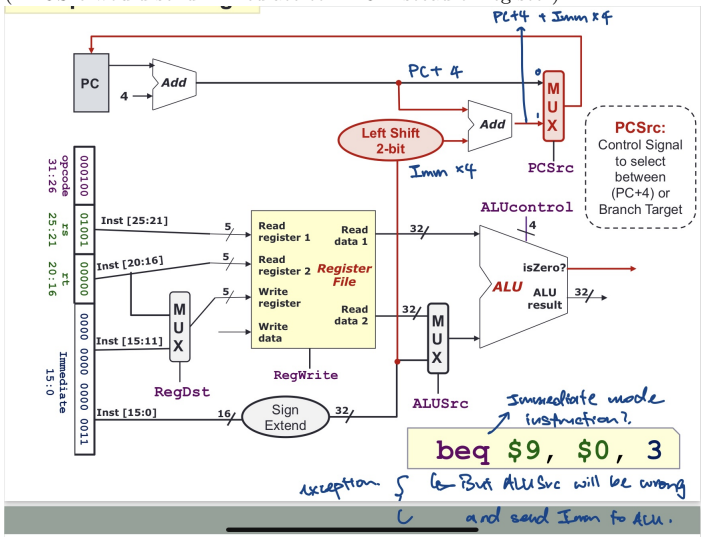


2. Instruction Decode
1. Read opcode to determine instruction type and field lengths
2. read data from necessary registers
- Register File: collection of 32 32-bit registers that can be read or written to
- Each instruction reads at most 2 registers and writes to at most 1 register
- `RegWrite` is the 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



3. ALU Stage aka EX stage
- 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

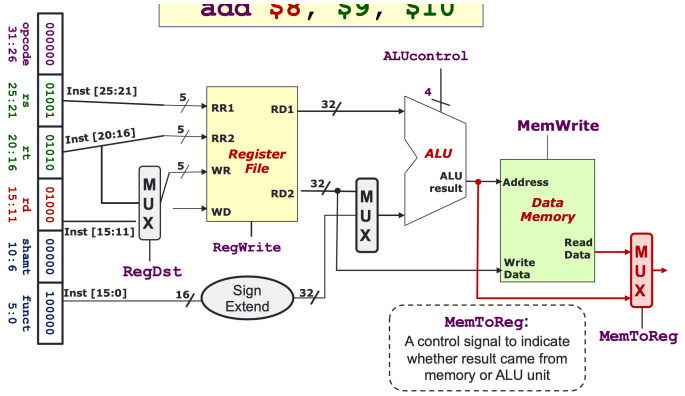
Only load and store instructions need to perform operations in this stage.
- Use memory address calculated by ALU stage
- Read or write to data memory

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

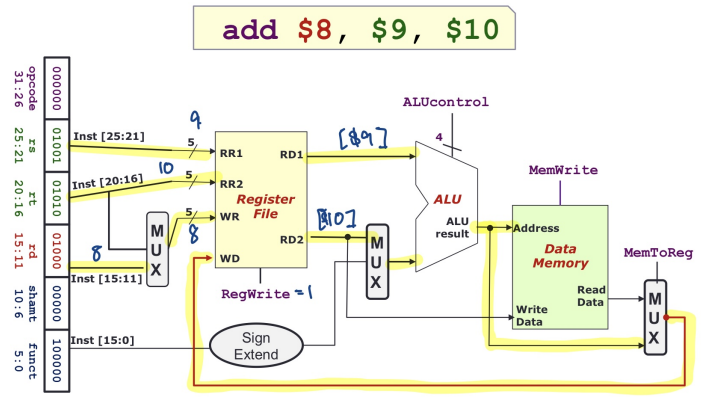
5. Register Write Stage

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

Just connect correct result to Register File Write Data input.

- Control: RegWrite, 1 write 0 nothing

5.5 Register Write Stage: **Routing**



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

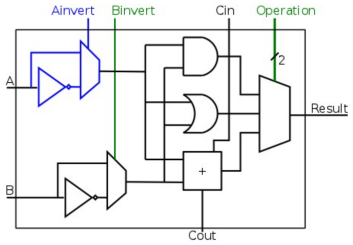
ALUControl

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?

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

- 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



Picture shows 1 "slice" of 32-bit ALU. Can think of actual ALU as 32 slices back to back

NOTE: `SUB` is implemented by inverting all bits in B, setting first C_{in} to 1

$$A + B' + 1 = A + \text{2s comp of } B$$

Multilevel Decoding

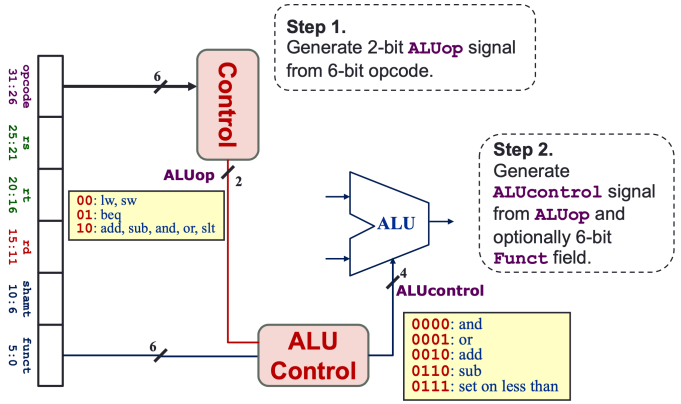
- Use some of the input to reduce cases, then generate full output
- Simplifies design process, reduce size of main controller, speedup circuit

Intermediate Signal: ALUop

- Use opcode to generate 2-bit `ALUop` signal

```
Instruction   |  ALUop  |  What
lw/sw         |  00     |  ADD
beq           |  01     |  SUB
R-type        |  10     |  depends on funct
```

Use `ALUop` signal and `funct` field from R-type to generate 4-bit `ALUControl`

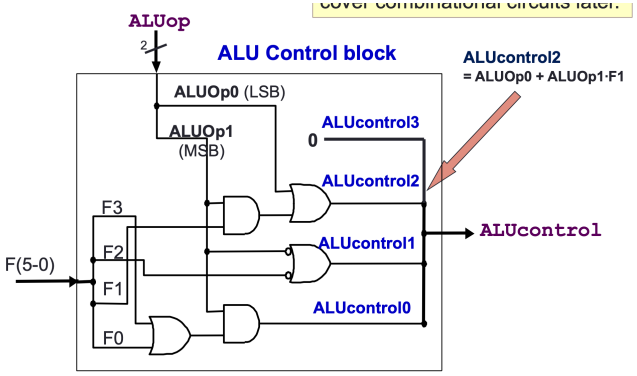


Opcode	ALUop	Instruction Operation	Funct field	ALU action	ALU control
lw	00	load word	xxxxxx	add	0010
sw	00	store word	xxxxxx	add	0010
beq	01	branch equal	xxxxxx	subtract	0110
R-type	10	add	10 0000	add	0010
R-type	10	subtract	10 0010	subtract	0110
R-type	10	AND	10 0100	AND	0000
R-type	10	OR	10 0101	OR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

- Table of Opcode, funct field and corresponding ALUop and ALUcontrol signals

	ALUop		Funct Field (F[5:0] == Inst[5:0])						ALU control
	MSB	LSB	F5	F4	F3	F2	F1	F0	
lw	0	0	X	X	X	X	X	X	0 0 1 0
sw	0	0	X	X	X	X	X	X	0 0 1 0
beq	0	1	X	X	X	X	X	X	0 1 1 0
add	1	0	1	0	0	0	0	0	0 0 1 0
sub	1	0	1	0	0	0	1	0	0 1 1 0
and	1	0	1	0	0	1	0	0	0 0 0 0
or	1	0	1	0	0	1	0	1	0 0 0 1
slt	1	0	1	0	1	0	1	0	0 1 1 1

- Table of how to generate each bit of ALUControl



- Circuit of ALUControl Block

$ALUControl0 = (ALUop_1 \cdot F_3) + (ALUop_1 \cdot F_0) = ALUop_1 \cdot (F_0 + F_3)$

$ALUControl1 = (ALUop_1 \cdot F_2)'$

$ALUControl2 = ALUop_0 + ALUop_1 \cdot F_1$

$ALUControl3 = 0$

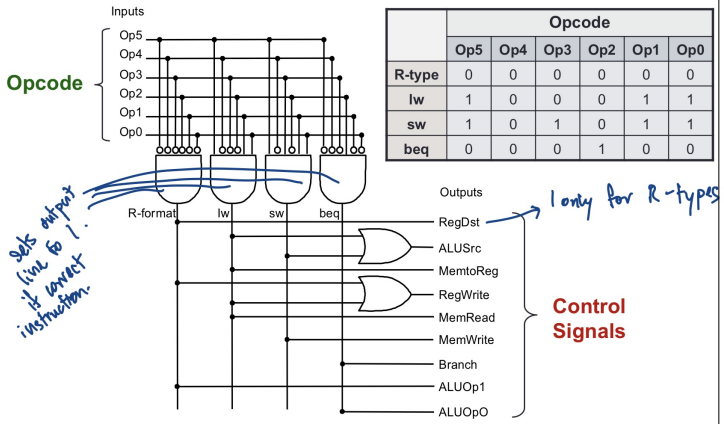
Control Design

	RegDst	ALUSrc	MemTo Reg	Reg Write	Mem Read	Mem Write	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

- Table of control signals for various instructions (outputs)

	Opcode (Op[5:0] == Inst[31:26])						
	Op5	Op4	Op3	Op2	Op1	Op0	Value in Hexadecimal
R-type	0	0	0	0	0	0	0
lw	1	0	0	0	1	1	23
sw	1	0	1	0	1	1	2B
beq	0	0	0	1	0	0	4

- Table of opcode for the various types of instructions (inputs)



	Opcode					
	Op5	Op4	Op3	Op2	Op1	Op0
R-type	0	0	0	0	0	0
lw	1	0	0	0	1	1
sw	1	0	1	0	1	1
beq	0	0	0	1	0	0

Instruction Execution

1. Read contents of register or memory
2. Perform computation
3. Write results

Perform all within a clock period to prevent reading a storage element when it is being written

Single Cycle Implementation

- All instructions take one cycle - Cycle must be at least as long as slowest instruction
- All instructions take as much time as slowest instruction

Multicycle Implementation

- Break up instructions into execution steps (IF, ID, EX, MEM, WR)
- Each execution step takes 1 clock cycle $\implies$ cycle time much shorter
- Cycle must be long enough to accomodate longest step
- Each step takes as much time as longest step
- Instructions take variable number of clock cycles to complete execution

Not covered

Pipelining

- One step per clock cycle
- Execute different steps of multiple instructions simultaneously

Boolean Algebra

Precedence of Operators

- Not (')
- And (.)
- Or (+)

Boolean Algebra Laws

Identity laws	
A + 0 = 0 + A = A	A · 1 = 1 · A = A
Inverse/complement laws	
A + A' = A' + A = 1	A · A' = A' · A = 0
Commutative laws	
A + B = B + A	A · B = B · A
Associative laws *	
A + (B + C) = (A + B) + C	A · (B · C) = (A · B) · C
Distributive laws	
A · (B + C) = (A · B) + (A · C)	A + (B · C) = (A + B) · (A + C)

Duality

If AND/OR operators and identity elements 0/1 in a Boolean Equation are interchanged, it remains valid

Theorems

Idempotency	
X + X = X	X · X = X
One element / Zero element	
X + 1 = 1 + X = 1	X · 0 = 0 · X = 0
Involution	
(X')' = X	
Absorption 1	
X + X·Y = X	X·(X + Y) = X
Absorption 2	
X + X'·Y = X + Y	X·(X' + Y) = X·Y
DeMorgans' (can be generalised to more than 2 variables)	
(X + Y)' = X' · Y'	(X · Y)' = X' + Y'
Consensus	
X·Y + X'·Z + Y·Z = X·Y + X'·Z	(X+Y)·(X'+Z)·(Y+Z) = (X+Y)·(X'+Z)

Standard Form

Every boolean expression can be expressed in SOP or POS.

- Literals: Boolean variable or complemented form, eg. `x or x'`
- Product Term: single literal or a product (AND) of several literals, eg `x or x.y.z'`
- Sum Term: single literal or a sum (OR) or several literals, eg `x or x+y+z'`
- Sum-of-Products (SOP): product term or sum (OR) of product terms, eg `x, x.y.z', x + y.z,`
- Product-of-Sum (POS): sum term or product (AND) of sum terms, eg `x, x+y+z', x.(x+y)`

Minterm and Maxterm

- Minterm of n variables is a **product term** that contains n literals from all the variables

eg. for two variables `x, y`, minterms are `x'.y', x'.y, x.y', x.y`

- Maxterm of n variables is a **sum term** that contains n literals from all the variables.

eg. for two variables `x, y`, maxterms are `x'+y', x'+y, x+y', x+y`

x	y	Minterms		Maxterms	
		Term	Notation	Term	Notation
0	0	x'·y'	m0	x+y	M0
0	1	x'·y	m1	x+y'	M1
1	0	x·y'	m2	x'+y	M2
1	1	x·y	m3	x'+y'	M3

Note: Each minterm is the compement of its corresponding maxterm, and vice-versa

Given truth table

- Obtain **sum of minterms** by gathering minterms (where output is 1)
- Obtain **product of maxterms** by gathering maxterms (where output is 0)

Conversion between standard forms

sum of minterm → product of max terms

-eg $F2 = \sum m(1, 4, 5, 6, 7) = \prod M(0, 2, 3)$

Logic Circuit

SOP expression can be implemented using
- 2-level AND-OR circuit
- 2-level NAND circuit

POS expression can be implemented using
- 2-level OR-AND circuit
- 2 level NOR circuit

Simplification

- Algebraic: using theorems
- K-map: no more than 6 variables
- Quine-McClusky: non-examinable

Gray Code

- Unweighted
- Only a **single bit change** from one code value to the next

Decimal	Binary	Gray Code	Decimal	Binary	Gray code
0	0000	0000	8	1000	1100
1	0001	0001	9	1001	1101
2	0010	0011	10	1010	1111
3	0011	0010	11	1011	1110
4	0100	0110	12	1100	1010
5	0101	0111	13	1101	1011
6	0110	0101	14	1110	1001
7	0111	0100	15	1111	1000

K-maps

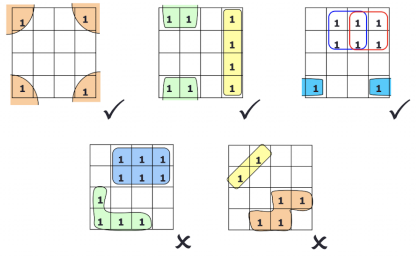
- Limited to 5 or 6 variables
- Each square represents a minterm
- Two adjacent squares represent minterms that differs by exactly 1 literal
- K-map of n variables has 2^n squares, each cell has n neighbors

Note: there is wrap-around in kmaps

How to use K-Map

- Group as many cells together as possible: larger group → less literals
- As few groups to cover all 1s: fewer groups → fewer product terms

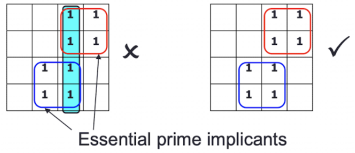
▪ Examples of valid and invalid groupings.



PI and EPI

Prime Implicant: product term obtained by combining the **maximum possible number of minterms** from adjacent squares, biggest group.

Essential Prime Implicant: A prime implicant that includes **at least one minterm not covered** by any other PIs.
Reduces number of redundant groups.

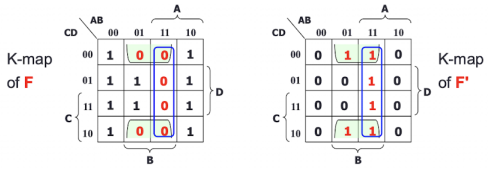


Algorithm for SOP

1. Find PIs
2. Find EPIs
3. Choose all EPIs
4. Choose PIs that cover minterms not covered by EPIs

Algorithm for POS

1. Same as SOP, but group maxterms (0s) instead



- This gives the SOP of F' to be
  F' = B·D' + A·B
- To get POS of F, we have
  F = (B' + D)·(A' + B')

Don't Care Conditions

- If output is not specified or invalid, it can be 0 or 1, and are denoted by X or d.
- Can be chosen to be either 1 or 0 to make a simpler SOP or POS expression.

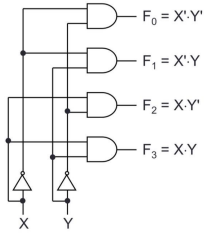
MSI Components

Medium Scale Integration

Decoder

Chooses one (of 2^n) output lines based on n input lines.
Each output corresponds to minterm of the n functions

X	Y	F_0	F_1	F_2	F_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1



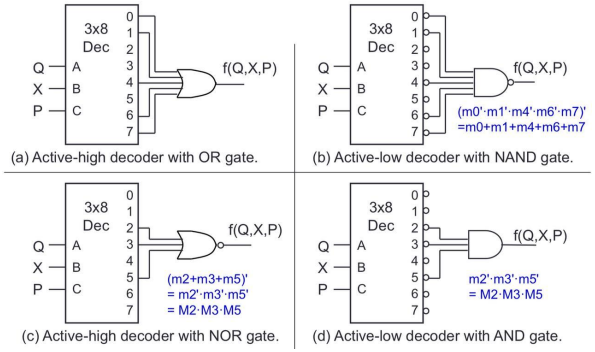
Decoder with enable
- output 0 when disabled, E = 0 in one enabled decoder

Active Low decoder
- selected line = 0, other lines 1
- implement function using NAND instead

Larger decoders can be constructed from smaller decoders and inverter using one of the inputs as enable signal

Implementing functions
- use decoder to generate minterms or maxterms
- inputs as selector lines to choose minterms or maxterms

(2/2) f(Q,X,P) = Σm(0,1,4,6,7) = ∏ M(2,3,5)



Encoder

- only one input = 1, returns the corresponding number of the input.
- Opposite of decoder

Priority Encoder
- Encoder output depends only on MSB, can have > 1 input line = 1

Demultiplexer

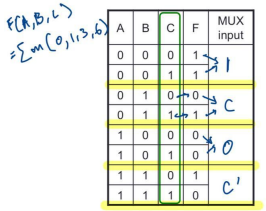
- Directs data from input line to one of the output lines, depending on the selector lines
- Circuit is same as decoder with enable, data input -¿ enable, selector lines -¿ decoder input

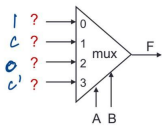
Multiplexer

Directs one of 2^n inputs to a single output line, using n selector lines. Known as a data selector
- output is the sum of (product of data lines and selection lines)
- $Output = I_0 \cdot (S_1' \cdot S_0') + I_1 \cdot (S_1' \cdot S_0) + I_2 \cdot (S_1 \cdot S_0') + I_3 \cdot (S_1 \cdot S_0)$
- $Output = I_0 \cdot m_0 + I_1 \cdot m_1 + I_2 \cdot m_2 + I_3 \cdot m_3$

Implementing Functions with Multiplexers
1. Express function in sum-on-minterm form
2. Connect n variables to n selector lines
3. Connect 1 to data line if it is a minterm of function or 0 otherwise

Note: Can use smaller multiplexers





Sequential Logic

Two types of sequential circuits
- Synchronous: outputs change only at specific time
- Asynchronous: outputs change at any time Multivibrator: a class of sequential circuits
- Bistable (2 stable states)
- Monostable or one-shot (1 stable state)
- Astable (no stable state)

Bistable Logic devices
- **Latches and Flip-Flops**
- Latches: pulse triggered
- Flip-flops: edge triggered

S-R Latch

Two inputs: S and R

Two complementary outputs: Q and Q'
- When $Q = 1$, in **SET** state
- When $Q = 0$, in **RESET** state

For Active High S-R Latch
- R = 1. S = 0: RESET
- R = 0, S = 1: SET
- R = 0, S = 0: No change
- R = 1, S = 1: invalid

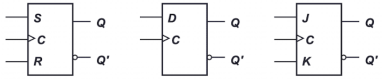
For Active Low S-R Latch
- flip

Gated Latches
- Enable input EN changes only if EN is HIGH

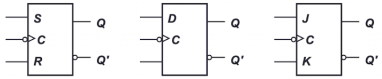
D Latch
- Same as SR Latch without invalid condition and only 1 input
- Single input D, D sent to S input, negated D sent to R input
- $Q^+ = D$

Flip Flops

Change state at rising or falling edge of clock signal



Positive edge-triggered flip-flops

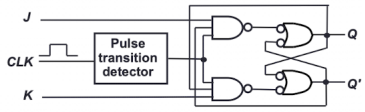


Negative edge-triggered flip-flops

J-K Flip-flop

Q and Q' are fed back to the pulse-steering NAND gates.
- No invalid state: when J and K are 1 → **Toggle**

- *J-K* flip-flop circuit:



- Characteristic table:

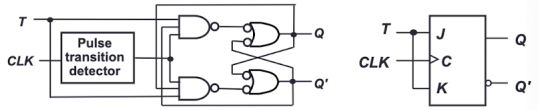
J	K	CLK	Q(t+1)	Comments
0	0	↑	Q(t)	No change
0	1	↑	0	Reset
1	0	↑	1	Set
1	1	↑	Q(t)'	Toggle

Q	J	K	Q(t+1)
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

Q(t+1) = ? $J \cdot Q' + K' \cdot Q$

T Flip-flop

Single input version of the J-K flip-flop, tie both input together



- Characteristic table:

T	CLK	Q(t+1)	Comments
0	↑	Q(t)	No change
1	↑	Q(t)'	Toggle

Q	T	Q(t+1)
0	0	0
0	1	1
1	0	1
1	1	0

Q(t+1) = ? $Q \oplus T$

Asynchronous Inputs

Affect state of flip-flop independent of clock
- $PRE = 1$, Q is **immediately** set to 1
- $CLR = 1$, Q is **immediately** set to 0
- $PRE = 0, CLR = 0$, normal flip-flop operation

Characteristic Table (for analysis)

J	K	Q(t+1)	Comments
0	0	Q(t)	No change
0	1	0	Reset
1	0	1	Set
1	1	Q(t)'	Toggle

S	R	Q(t+1)	Comments
0	0	Q(t)	No change
0	1	0	Reset
1	0	1	Set
1	1	?	Unpredictable

D	Q(t+1)	
0	0	Reset
1	1	Set

T	Q(t+1)	
0	Q(t)	No change
1	Q(t)'	Toggle

Analysis of Sequential Circuit

1. Derive state equations / input equations from ff inputs, and output functions (if any)
2. Derive state table and hence diagram, using state equations and characteristic table

Excitation Table (for Design)

Excitation Tables: given the required transition from present state to next state, determine the flip-flop inputs

Q	Q^+	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

JK Flip-flop

Q	Q^+	S	R
0	0	0	X
0	1	1	0
1	0	0	1
1	1	X	0

SR Flip-flop

Q	Q^+	D
0	0	0
0	1	1
1	0	0
1	1	1

D Flip-flop

Q	Q^+	T
0	0	0
0	1	1
1	0	1
1	1	0

T Flip-flop

Design Procedure
1. Start with specs - state diagram or state table.
2. Derive state table
3. Perform state reduction if necessary
4. Perform state assignment
5. Determine number of flip-flops and label them.
6. Choose type of flip-flop
7. Derive Circuit excitation and output tables from the state table
8. Derive circuit output functions and flip-flop input functions (simplify w kmap if necessary)
9. draw logic diagram

Pipelining

Pipelining doesn't help latency of single task, but helps throughput of entire workload

Multiple tasks operating simultaneously using different resources

Delays
- limited by slowest pipeline stage
- stall for dependencies

MIPS Pipeline

Five pipelining stages
- **IF**: Instruction Fetch
- **ID**: Instruction Decode and Register Read
- **EX**: Execute operation or calc address (ALU)
- **MEM**: Read/Write from/to memory
- **WB**: Write back the result into a register

Each execution stage takes 1 clock cycle

General flow of data is from one stage to the next

Exception: Update of PC and write back to reg file

Pipeline Datapath

Data used by **same instruction** in later pipeline stages need to be stored somewhere (due to multicycle(?))
- Additional registers in datapath called **pipeline registers**
- `IF / ID`: Register between `IF` and `ID`
- `ID / EX`: Register between `ID` and `EX`
- `EX / MEM`: Register between `EX` and `MEM`
- `MEM / WB`: Register between `MEM` and `WB`

No need for register at end of `WB` stage as it is the end of the command.

Note: Need to send the "Write Register" number along through the pipeline

registers so that it can be used in WB stage

Pipeline Control

Difference from single-cycle datapath: Each control signal belongs to a particular pipeline stage

	EX Stage				MEM Stage			WB Stage	
	RegDst	ALUSrc	ALUop		Mem Read	Mem Write	Branch	MemTo Reg	Reg Write
			op1	op0					
R-type	1	0	1	0	0	0	0	0	1
lw	0	1	0	0	1	0	0	1	1
sw	X	1	0	0	0	1	0	X	0
beq	X	0	0	1	0	0	1	X	0

Number of cycles needed in Idea pipeline, with I instructions:
$I + N - 1$ in an N cycle pipeline (need $N - 1$ cycles to fill pipeline)

Pipeline processor can gain N times speedup, where N is the number of pipeline stages (ideal pipeline)

Pipeline Hazards

Problems that prevent next intruction from immediately following previous instruction
1. Structural hazards: Simultaneous use of hardware
2. Data hazards: Data dependencies between instructions
3. Control hazards: Change in program flow

2 and 3 are Instruction Dependencies (instructions have relationships that prevent pipeline execution)

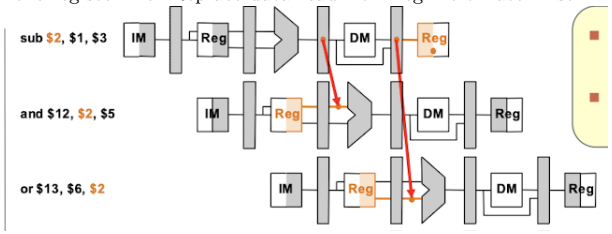
Structural Hazards

1. IF and MEM both access memory, clash between Inst 1 and 3
Solution: Split memory into Data Memory and Instruction Memory.

2. ID and WB both use reg file
Solution: Registers are fast; Split cycle into two, write in first half, read in second half

Data dependency: RAW

Read-After-Write: Occurs when a later instruction **reads** from the destination register **writter** by an earlier instruction
Solution: **Forward** the result to any later instruction before it is reflected in the register file. Replace data read from reg file of later inst.



Exception: lw cannot be fully solved with forwarding. Data is produced in MEM. Must stall 1 cycle
In general, without data forwarding stall **2** cycles (even if lw is preceding).

Control Dependency

Definition: An instruction j is control dependent on i if i controls whether or not j executes. (branch inst)

Problem: Branch Decision is made at MEM stage.
Solution:
1. Stall the pipeline for 3 cycles.
2. Early Branching: move the branch decision to an earlier stage
3. Branch Prediction: Guess outcome of branch
4. Delayed Branching: Do something useful while waiting for outcome

Early Branching

Make decision in ID stage instead of MEM
- Move branch target address calculation
- Move register comparison → cannot use ALU for register comparison any more
- Still need to stall 1 cycle

NOTE if register involved in comparison is produced by preceding instruction (RAW), need to stall **extra 1** cycle (total 2 cycles)
If preceding instruction is **lw**, need to stall **extra 2** cycles → back to 3 cycles of delay

Branch Prediction

Simplest prediction scheme: branch not taken
Not taken: Guessed correctly → No pipeline stall
Taken: Guessed wrongly → Wrong instructions in the pipeline → **Flush** successor instruction from the pipeline

Delayed Branching

Move **non-control dependent instructions** into the X slots following a branch (1 if early branching)
- These instructions are executed **regardless** of branch outcome
- Must have the same behaviour as original code **NOTE**: Unless branch prediction is used in which case wrong instruction **will be flushed!**

Best Case Scenario:
There is an instruction **preceding the branch** which **can be moved** into the delayed slot.
- Program correctness must be preserved

Worst Case Scenario No instruction can be found - no-op instruction instead (nop)

Cache

small but fast SRAM near CPU, makes slow main memory appear faster

Memory Hierarchy

Registers are in the processor, if operands are in memory, need to load them into processor, operate, store back.
- Fast RAM -¿ expensive
- Cheap RAM -¿ slow

Therefore have small but fast memory near CPU, large but slow memory farther from CPU
Keep frequently and recently used data in **smaller but faster** memory

Principle of Locality

Program accesses only a small portion of the memory address space within a small time interval
Types of Locality
1. Temporal Locality - If an item is referenced, it will tend to be referenced soon again
2. Spatial Locality - If an item is referenced, nearby items will tend to be referenced soon

Different locality for instruction and data

Terminology

- Hit: Data is in the cache
- Hit rate: Fraction of memory accesses that hit
- Hit time: Time to access cache
- Miss: Data is not in cache
- Miss Rate: 1 - Hit Rate
- Miss Penalty: Time to replace cache block + Hit Time

Hit time ¡ Miss penalty
Average Access Time = Hit rate x Hit time + (1 - Hit rate) x Miss penalty

Memory to Cache Mapping

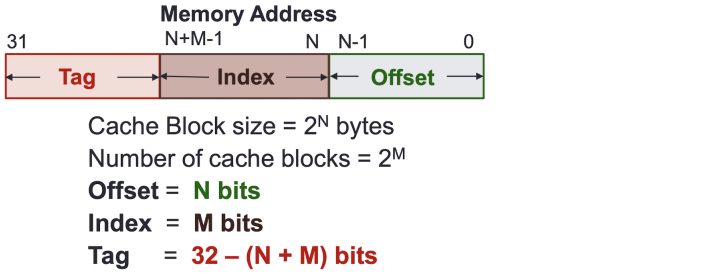
Cache Block: Unit of transfer between memory and cache
- Typically 1 or more words (spatial locality)

Observations:
1. 2^N-byte blocks are aligned at 2^N-byte boundaries
2. Addresses of words within a 2^N-byte block have identical $(32 - N)$ MSB.
3. Bits $[31 : N]$ → **block number**
4. Bits $[N - 1 : 0]$ → **byte offset** within a block

Direct Mapped Cache

- Each Block has exactly one location to be mapped to in cache (known as the **cache index**)
- Multiple blocks may be mapped to the same location in cache
- But each is uniquely identified by its tag

Tag = block number / no. of cache blocks.



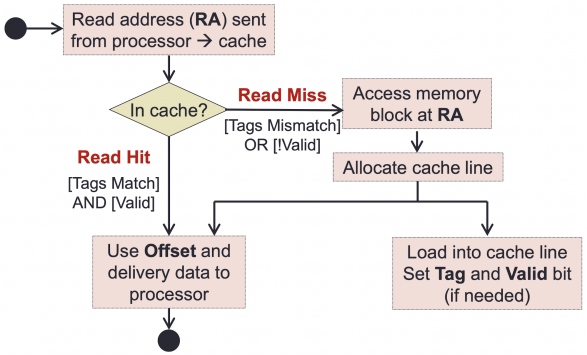
Note: For MIPS, the 2 LSB of the offset is byte number, next 2 bit are word number within the block.
Word number within block tells you which words to bring into cache

Cache Structure

- Data block
- Tag
- Valid bit

Reading Data

Cache hit when Tag[index] = Tag[Memory Address] **and** Valid bit = 1
Note: initially all valid bits = 0



Writing Data

Changing only memory or cache → cache and memory out of sync

Two policies:

1. Write-through cache: Write data to both cache and memory
2. Write-back cache: Only write to cache, write to main memory when cache block is evicted

Write Through Cache

Problems: Writes slowly at the speed of main memory

Solution: Put a write buffer between cache and memory; CPU write to cache and buffer, mem controller write to memory

Write Back Cache

Problem: Wasteful to write back every block that is evicted

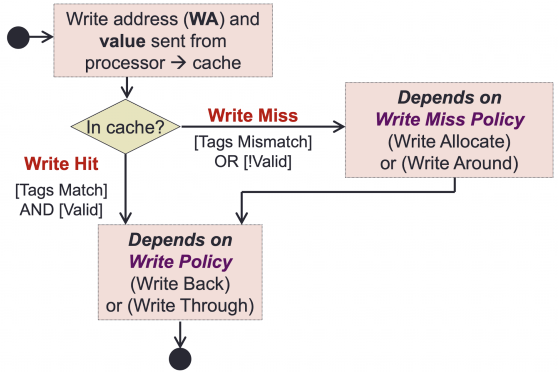
Solution: Add a 'Dirty Bit' to each block and only write to memory if dirty bit = 1

Types of Misses

1. Compulsory / Cold Misses: On the first access to a block; the block must be brought into the cache
2. Conflict / Collision Misses: Occurs in SA cache and Direct Mapped cache; when several blocks are mapped to the same block/set
3. Capacity Misses: Occurs when blocks are discarded from cache as cache cannot contain all blocks needed

Handling Cache Misses

- Read Miss: load block into cache, read from cache
- Write Miss: Write Allocate - Load block into cache, change word, write to main memory depending on write policy
- Write Miss: Write Around - Write **straight to main mem** without loading block into cache



Block Size

Reduces cache miss rate (due to spatial locality) up to a point, then increases (due to lower number of blocks in cache)

Set Associative Cache

Solution to the problem that there are too many conflicts

N-way Set Associative Cache - Memory block can be placed in N locations in cache

In 2-way SA cache

- Cache is split into sets, each set contains 2 blocks.
- Memory block is mapped to a unique set, and can be placed in either block → need to search **both blocks** when searching cache

Cache Block size = 2^N bytes
Number of cache sets = 2^M
Offset = N bits
Set Index = M bits
Tag = 32 – (N + M) bits

Observation: It is essentially unchanged from the direct-mapping formula

Note: Formula is similar to DM cache, but **Set Index** rather than **Index**

Advantage of Associativity

Rule of Thumb - A direct mapped cache of size N has about the same miss rate as a 2-way SA cache of size N/2

Fully Associative Cache

Memory block can be placed in any location in cache

Pros: Can be placed in any location in cache

Cons: Need to search all of the locations in cache for a block

Cache Block size = 2^N bytes
Number of cache blocks = 2^M
Offset = N bits
Tag = 32 – N bits

Observation: The block number serves as the tag in FA cache.

Note: No conflict misses but have capacity misses

Cache Performance

1. Compulsory/Cold misses remain same irrespective of cache size and associativity
2. Conflict Misses decrease with increasing associativity
3. Conflict Misses = 0 for FA cache
4. Capacity Misses decrease with cache size

Block Replacement Policy (for SA and FA cache)

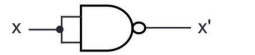
Least Recently Used (LRU) policy:

- How: for each cache hit, record the cache block that was accessed. When replacing, choose the block that has not been accessed for a long time
- Why: Temporal Locality
- Note: need a supporting Data Structure to keep track of LRU

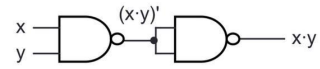
Drawback: Hard to keep track if there are many choices

Other replacement policies

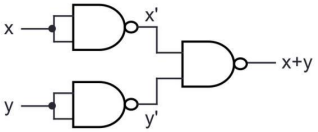
- First in First out
- Random Replacement
- Least Frequently Used



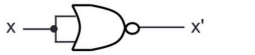
$(x \cdot x)' = x'$ (idempotency)



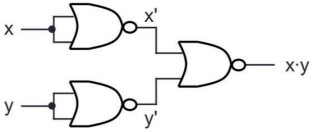
$((x \cdot y)' \cdot (x \cdot y)')' = ((x \cdot y)')'$ (idempotency)
$= x \cdot y$ (involution)



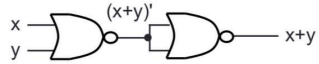
$((x \cdot x)' \cdot (y \cdot y)')' = (x' \cdot y')'$ (idempotency)
$= (x')' + (y')'$ (DeMorgan)
$= x + y$ (involution)



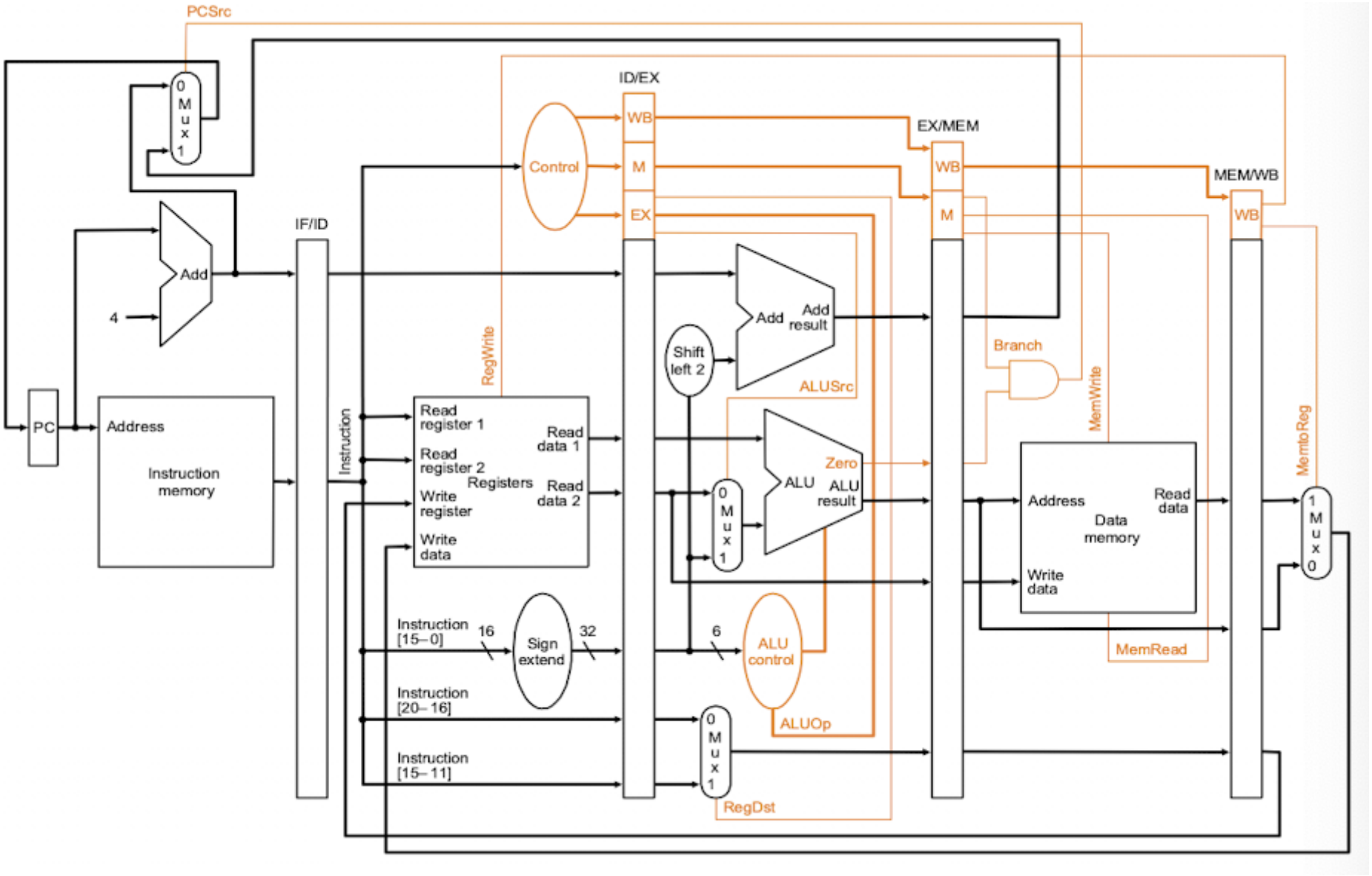
$(x+x)' = x'$ (idempotency)



$((x+x)'+(y+y)')' = (x'+y')'$ (idempotency)
$= (x')' \cdot (y')'$ (DeMorgan)
$= x \cdot y$ (involution)



$((x+y)'+(x+y)')' = ((x+y)')'$ (idempotency)
$= x + y$ (involution)


PCSrc
0
M
u
x
1
ID/EX
WB
M
EX
Control
EX/MEM
WB
M
MEM/WB
WB
IF/ID
Add
4
PC
Address
Instruction memory
Instruction
RegWrite
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Shift left 2
Add
Add result
ALUSrc
Branch
MemWrite
MemtoReg
ALU
Zero
ALU result
Address
Data memory
Read data
Write data
MemRead
Instruction [15–0]
16
Sign extend
32
6
ALU control
ALUOp
Instruction [20–16]
Instruction [15–11]
RegDst