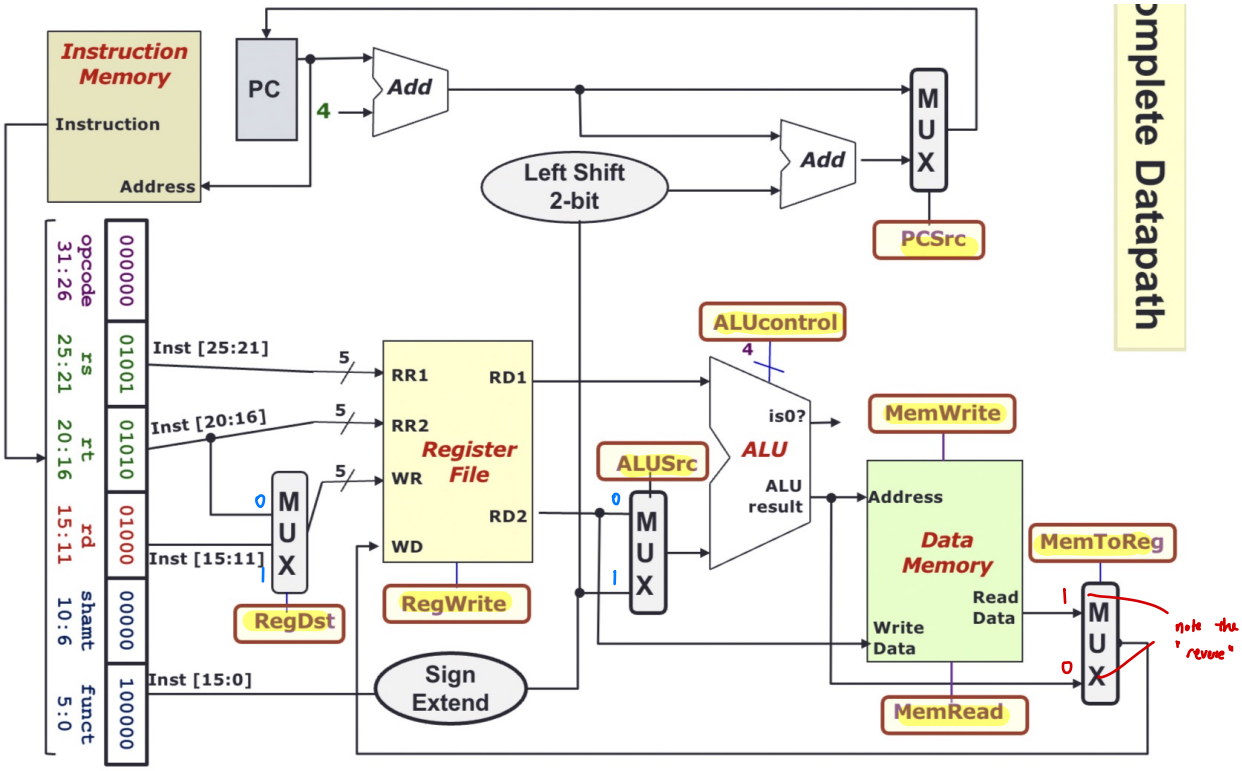
Control Path (in Processor)

- how control signals are generated.



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

R, I, J- format

- Control signals are generated based on the instruction to be executed:
  - use opcode to differentiate instruction format
  - eg. R-format, RegDst = 1 because of rd register
  - R-type instruction has additional info:
    - The 6-bit "funct" field

- Idea:

  output is immediately affected by input.

  Design a combinational circuit to generate three signals based on Opcode and possibly function code (R-format)

  ⇒ a control unit is needed

A, B → CC → o/p vs sequential circuit

A, B → SC → o/p, clock

- output changes depending on clock signal and with input
- eg. PC - val changes at "rising clock edge"

RegDst

- False (0): Write register = rt [20:16].
- True (1): Write register = rd [15:11].

RegWrite

- False (0): No register write
- True (1): New value will be written.

  WD → WR

  eg. WD = 110, WR = $7

  if RegWrite = 1, [$7] = 110

  else, nothing.

ALUSrc

- False (0): Operand 2 = Register Read Data 2 (RD2)
- True (1): Operand 2 = SignExt(Immed [15:0])

MemRead (lw)

- False (0): Nothing.
- True (1): Read memory using address

MemWrite (sw)

- False (0): Nothing.
- True (1): Memory[address] ← RD2 (From Write Data)

MemToReg

- True (1): Register write data = Memory read data (lw)
- False (0): Register write data = ALU result.

PCSrc

- False (0): PC = PC + 4

  All other instr.

- True (1): PC = SignExt([15:0]) << 2 * (PC + 4)

opcode

When branch inst / taken

eg. beq $s0, $s1, label

equal?

isO output from ALU

1 (same) 0 (otherwise)

⇒ PCSrc = Branch AND isZero.

⇒ use AND gate

control (instr.)

isO? (ALU)

PCSrc

- Able to be generated with opcode directly.

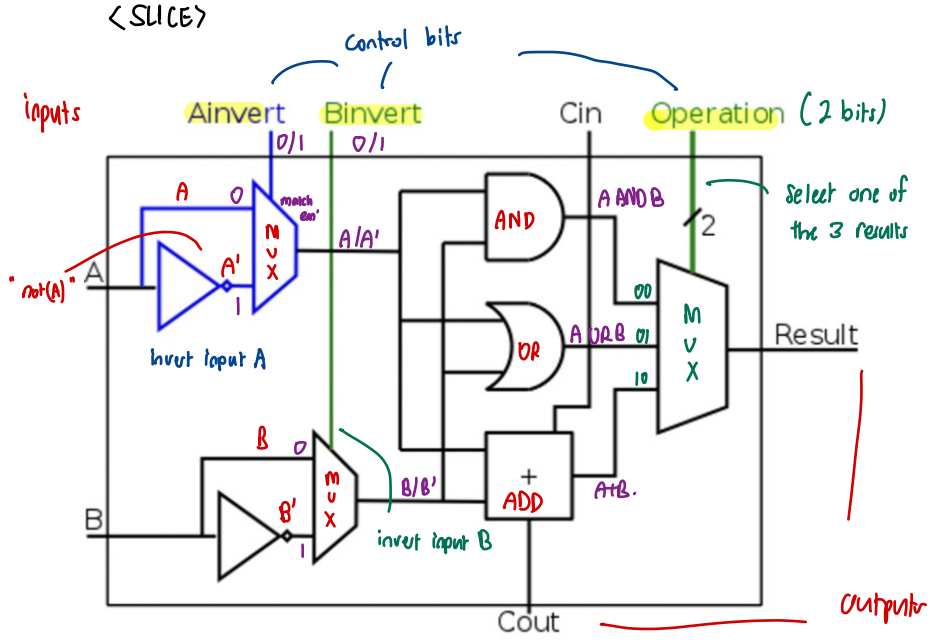
ALU Control

- 4 bits control signal.

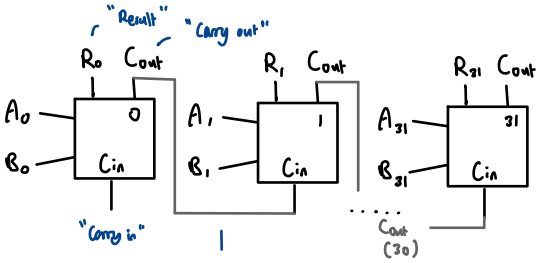
ALUcontrol	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	slt
1100	NOR

- ALU is a combinational circuit.
- A 1-bit MIPS ALU: (32 bits in total)

<SLICE>



i.e.



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

A' AND B'
→ De morgan's law
→ NOT (A OR B)
→ A NOR B

A+B',
Cin = 1
2's comp.
⇒ A+B'+1
A − B.

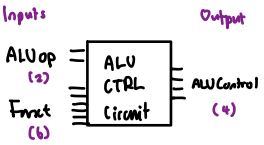
Design of ALUcontrol signal

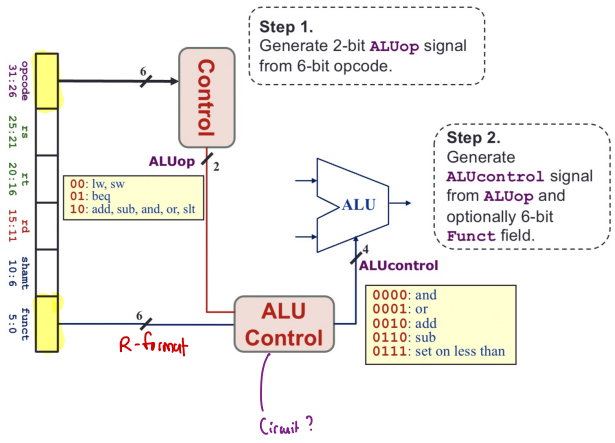
- Truth Table ⇒ Simplified Boolean Expression.
- depends on opcode and function code
  - 12 variables (bits)
  - 2^{12} space for control signal
  - If Brute force, will result in a very complicated circuit.
- Multilevel Decoding approach
  - Use some of the input to reduce the cases, then generate the full output.
  - Simplify design, reduce size of main controller, speedup circuit.
- New signal: ALUop
  - use opcode to generate a 2 bit ALUop signal

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

(op in ALU; 00 — add; 01 — sub; 10 — funct code.)

  - Use ALUop and function code (for R-type) to generate ALU control.





Opcode	ALUop (2)	Instruction Operation	Funct field (6)	ALU action	ALU control
lw	00	load word	xxxxxx (don't care term)	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

Don't care about Funct

inputs

output

Generation of 2-bit **ALUop** signal will be discussed later

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

generated from

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

- Truth Table

from opcode

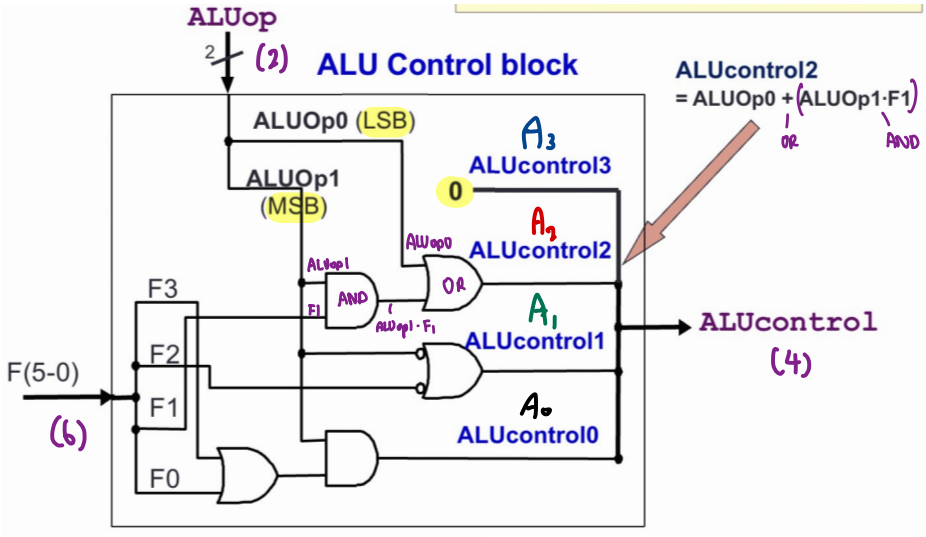
	ALUop		Funct Field ("All 1s" "All 0s" (F[5:0] == Inst[5:0])						ALU control
	$ALUop_1$ MSB	$ALUop_0$ LSB	F5	F4	F3	F2	F1	F0	A_3 A_2 A_1 A_0
lw	0	0	x	x	x	x	x	x	0010
sw	0	0	x	x	x	x	x	x	0010
beq	~~0~~ x	1	x	x	x	x	x	x	0110
add	1	~~0~~ x	~~x~~ x	~~0~~ x	0	0	0	0	0010
sub	1	~~0~~ x	~~x~~ x	~~0~~ x	0	0	1	0	0110
and	1	~~0~~ x	~~x~~ x	~~0~~ x	0	1	0	0	0000
or	1	~~0~~ x	~~x~~ x	~~0~~ x	0	1	0	1	0001
slt	1	~~0~~ x	~~x~~ x	~~0~~ x	1	0	1	0	0111

$ALUcontrol3 = 0$

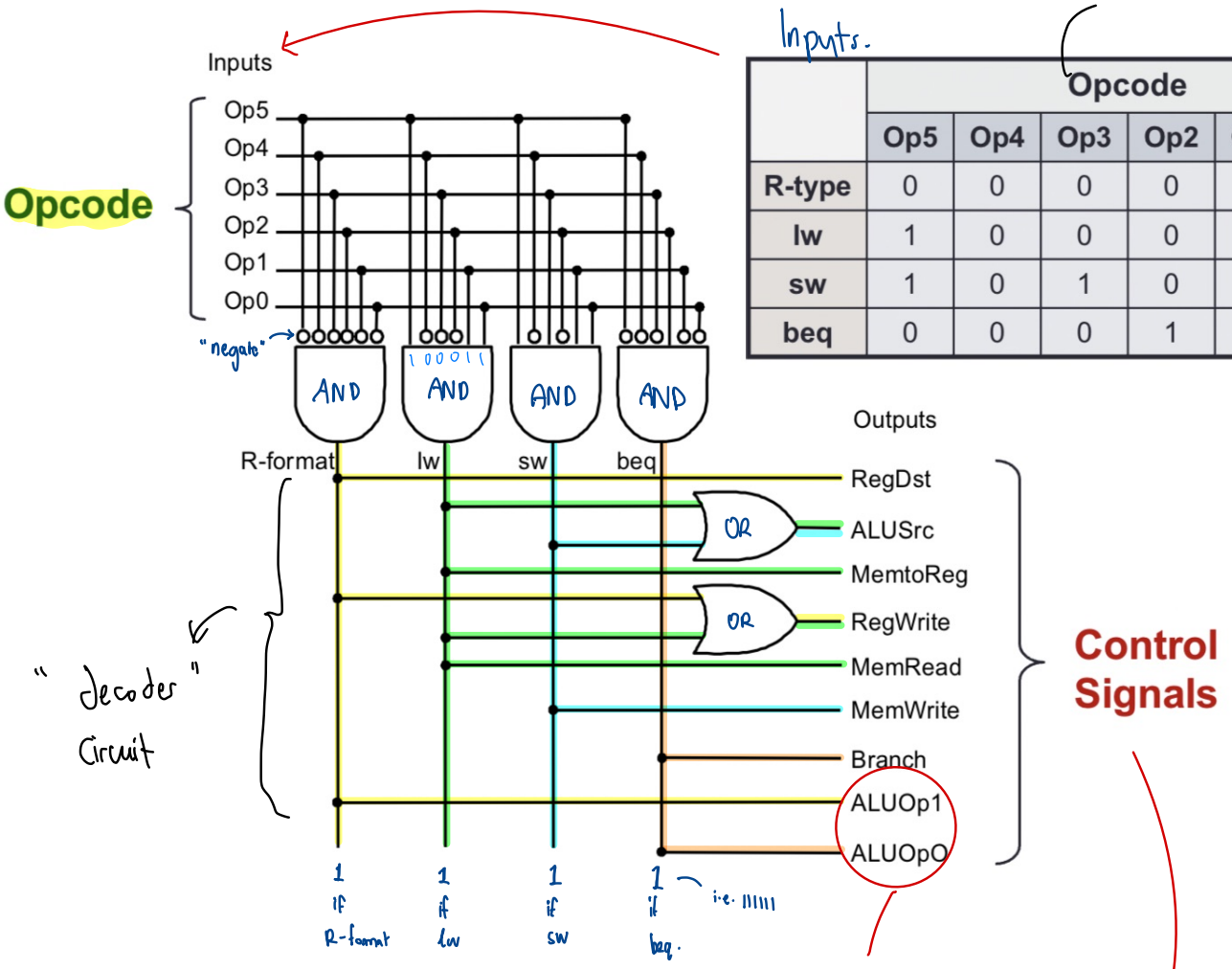
$ALUcontrol2 = 1$ if $(ALUop0)$ OR $(ALUop1$ AND $F_1)$ — equals 1

under logic design

- Simplify the process!
- Derive an expression for $A_3 A_2 A_1 A_0$
- Simple Combinational logic



- Circuit



	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

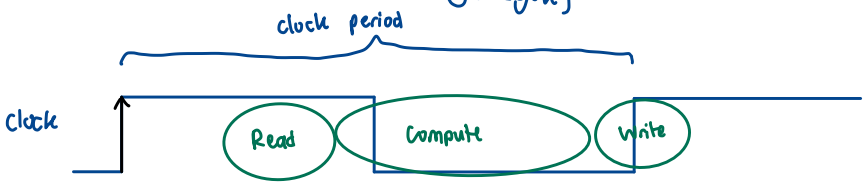
- Outputs

	RegDst	ALUSrc	MemTo Reg	Reg Write	Mem Read	Mem Write	Branch	ALUop	
								op1	op0
R-type	1	0 reg	0	1	0	0	0	1	0
lw	0 RT	1 immd	1	1	1	0	0	0	0
sw	X	1 immd	X	0	0	1	0	0	0
beq	X	0 reg	X	0	0	0	1	0	1

Instruction Execution

1. Read content of one or more storage elements (register/memory)
2. Perform computation through some combinational logic
3. Write results to one or more storage elements (register/memory)

- All performed within a clock period. (Single Cycle)



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

- Drawback:

memory (2ns), ALU/adder (2ns), register file access (1ns)

Instruction	Inst Mem	Reg read	ALU	Data Mem	Reg write	Total
ALU	2	1	2	X	1	6
lw	2	1	2	2	1	8
sw	2	1	2	2	X	7
beq	2	1	2	X	X	5

use longest instr.'s clock cycle.
⇒ other instr. as slow as the slowest one.

- Solution?

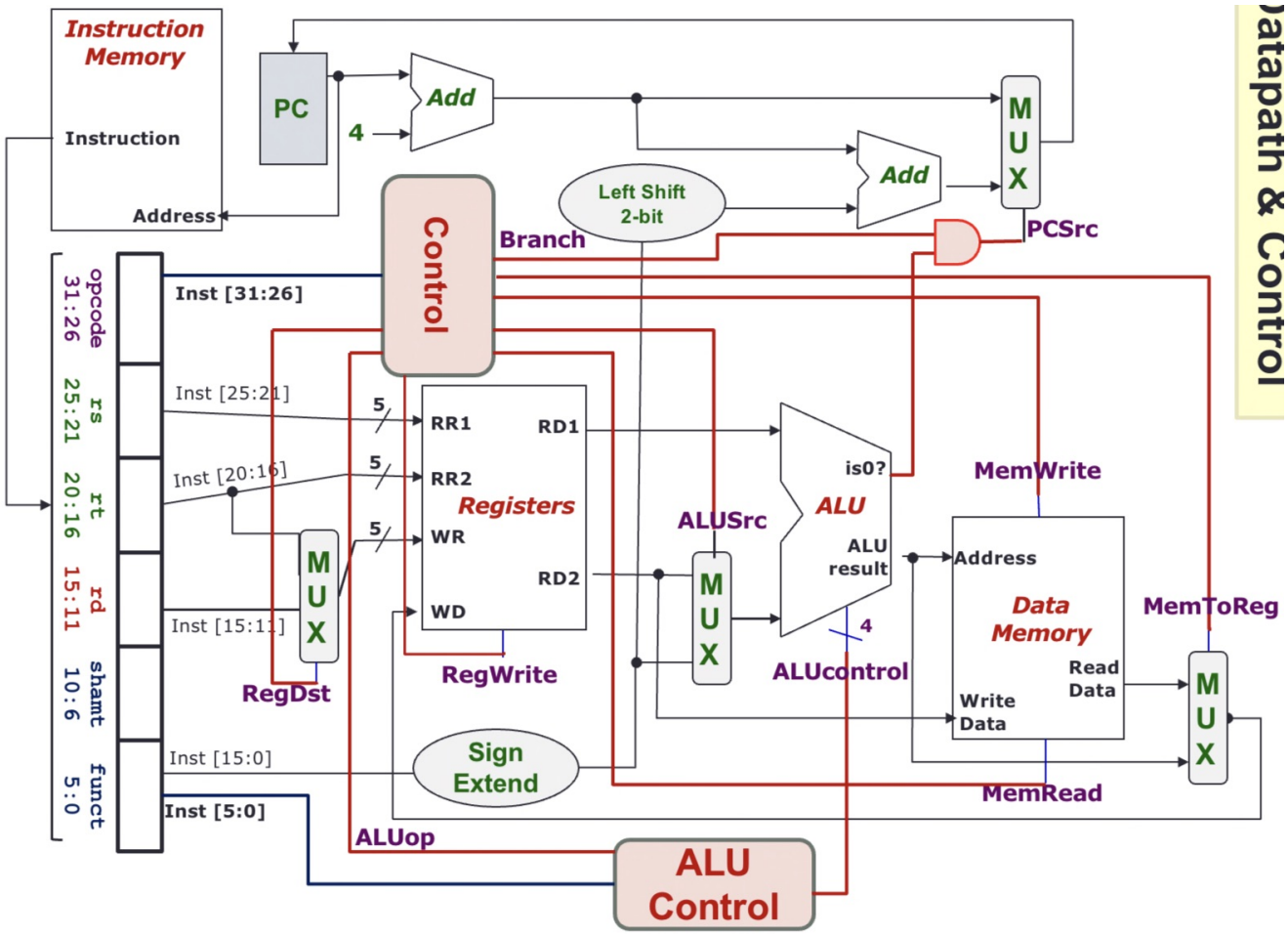
6. Solution #1: Multicycle Implementation

- Break up the instructions into execution steps:
  1. Instruction fetch
  2. Instruction decode and register read
  3. ALU operation
  4. Memory read/write
  5. Register write
- Each execution step **takes one clock cycle**
  - ➔ **Cycle time is much shorter, i.e., clock frequency is much higher**
- Instructions take variable number of clock cycles to complete execution
- Not covered in class:
  - See Section 5.5 of COD if interested

But each step will take longest step's clock cycle
∴ might be slower.

Pipelining

- Break up instr. into execution steps, one per clock cycle.
- Allows different instr. to be in different execution steps simultaneously.



Instruction	Opcode	Sh_B	S_zext	Regwrite	Regdst	Alusrc	Branch	Brchne	Blez	Bltz	Bgtz	Memwrite	Memtoreg	Jump	Jal	ALUop	hlt
R_type	000000	xx	x	1	01	00	0	0	0	0	0	0	00	0	0	110	0
lw	100011	xx	x	1	00	01	0	0	0	0	0	0	01	0	0	000	0
sw	101011	11	x	0	xx	01	0	0	0	0	0	1	xx	0	0	000	0
beq	000100	xx	x	0	xx	00	1	0	0	0	0	0	xx	0	0	001	0
bne	000101	xx	x	0	xx	00	0	1	0	0	0	0	xx	0	0	001	0
blez	000111	xx	x	0	xx	00	0	0	1	0	0	0	xx	0	0	001	0
bltz	000001	xx	x	0	xx	00	0	0	0	1	0	0	xx	0	0	001	0
bgtz	000110	xx	x	0	xx	00	0	0	0	0	1	0	xx	0	0	001	0
addi	001000	xx	x	1	00	01	0	0	0	0	0	0	00	0	0	000	0
addiu	001001	xx	x	1	00	01	0	0	0	0	0	0	00	0	0	000	0
j	000010	xx	x	0	xx	xx	x	x	x	x	x	0	xx	1	0	xxx	0
jal	000011	xx	x	1	10	xx	x	x	x	x	x	0	xx	1	1	xxx	0
andi	001100	xx	x	1	00	10	0	0	0	0	0	0	00	0	0	010	0
ori	001101	xx	x	1	00	10	0	0	0	0	0	0	00	0	0	011	0
xori	001110	xx	x	1	00	10	0	0	0	0	0	0	00	0	0	100	0
slti	001010	xx	x	1	00	01	0	0	0	0	0	0	00	0	0	101	0
sltiu	001011	xx	x	1	00	01	0	0	0	0	0	0	00	0	0	101	0
lui	001111	xx	x	1	00	11	0	0	0	0	0	0	00	0	0	000	0
lb	100000	xx	0	1	00	01	0	0	0	0	0	0	11	0	0	000	0
lbu	100100	xx	1	1	00	01	0	0	0	0	0	0	11	0	0	000	0
lh	100001	xx	0	1	00	01	0	0	0	0	0	0	10	0	0	000	0
lhu	100101	xx	1	1	00	01	0	0	0	0	0	0	10	0	0	000	0
sb	101000	00	x	0	xx	01	0	0	0	0	0	1	xx	0	0	000	0
sh	101001	01	x	0	xx	01	0	0	0	0	0	1	xx	0	0	000	0
hlt	111100	xx	x	0	xx	xx	x	x	x	X	x	0	xx	x	x	xxx	1