Digital System Design

Hardware Implementation of Single Cycle RISC-V

Speaker: Ya-Cheng Ko

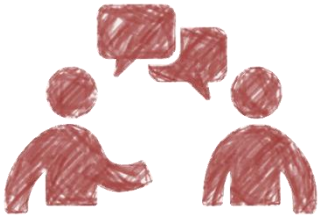
Instructor: 吳安宇教授

Date: 2020/04/16

Single-cycle RISCV Processor Supplementary Material

Instruction Set Architecture

- The interface between hardware and software
  - Complexed Instruction Set Computer(CISC)
  - Reduced Instruction Set Computer(RISC)



Comparison	CISC (early)	RISC
Instruction count	>200	<100
Instruction length	Not fixed	Fixed
Area	High	**Low**
Timing	High	**Low**
Program size	**Small**	Large
Instruction cycle	Multiple cycle	Single cycle
Celebrity	x86 (PC)	ARM (Mobile)
Description	Execute 20% instruction in 80% time	As simple as possible

Born of Hope: RISC-V

- Birth: UC Berkeley, Professor Krste Asanovic, 2010
- RISC-V Foundation in 2015
  - Comprises more than 100 member organizations, collaborative community of software and hardware innovators.
- Goals
  - Royalty-free for any purpose
  - Become a standard ISA that connect well in hardware and software.
- Design Concept
  - As Simple As Possible
  - Modularize function blocks
- Award
  - The Linley Group's Analyst's Choice Award for Best Technology (The instruction set, 2017) RISCV


Berkeley Architecture Research
DRAPER
bluespec
NVIDIA
DOVER MICROSYSTEMS
Hewlett Packard Enterprise
Mellanox TECHNOLOGIES
Microsoft
NXP
C-SKY
HUAWEI
cortus
Micron
SAMSUNG
SiFive
Rambus
Cryptography Research
ORACLE
IBM
Western Digital
Google
Microsemi
QUALCOMM
RISC-V Foundation: 65+ Members
ESPRESSIF
MEDIATEK
AMD
ANDES TECHNOLOGY
IDT
SEGGER It simply works!
BAE SYSTEMS
Esperanto Technologies
VeriSilicon
LATTICE SEMICONDUCTOR
GRAY RESEARCH
runtime.io
antmicro
codasip
GRIDOOG
SHC SH CONSULTING
INTRINSIX
ETH zürich
TRINAMIC
imperas
ISTUARY
GREENWAVES
PROCESSOR
MIT CSAIL
ROA LOGIC
Rumble Development
lowRISC
UBILITE Hello IoT World
ICT
EMDALO TECHNOLOGIES
EMBECOSM
Sur Technology
Syntacore
Blockstream
VectorBlox
Technolution
ultraSoC
csem
PRINCETON UNIVERSITY
BERKELEY LAB

Basic Design of RISC-V

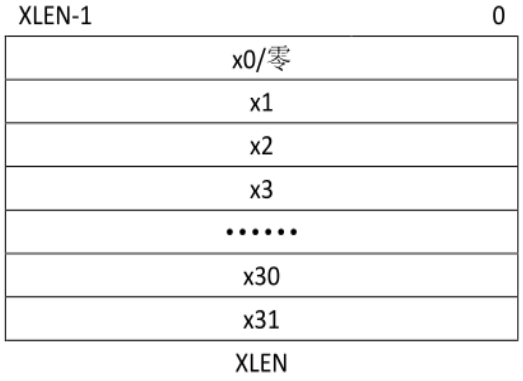
- Base Integer Instruction Set Architecture (ISA)
  - must be present in any implementation

Name	Number	Description
RV32I	47	Including arithmetic, branch, store/load. 32 bits user address space, 32*32-bits registers
RV32E	47	subset of RV32E, 16*32-bits registers ⟸ **Low power**
RV64I	59	64 bits user address space, 32*64-bits registers
RV128I	71	128bits user address space, 32*128-bits registers

- Standard Extension **(Modular design)**

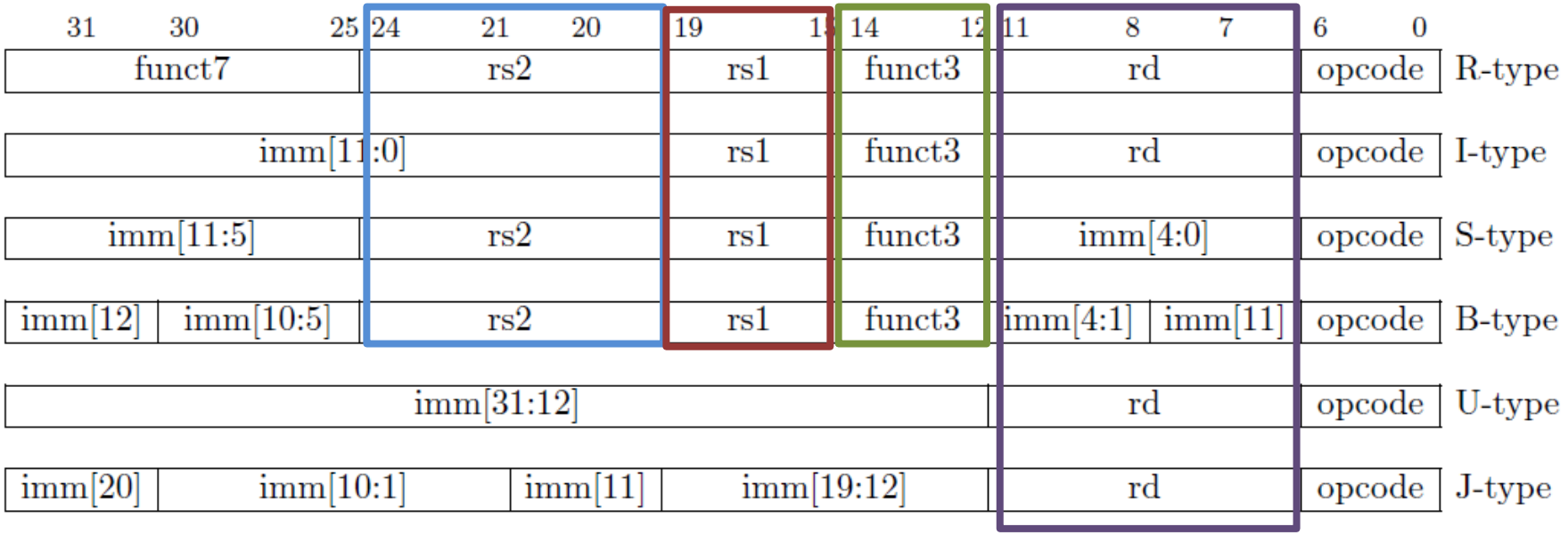
Name	Number	Description
C	**53**	**16bits compressed instruction**
M	8	Multiplication, division, mod
F	26	Floating type instruction
D	26	Double type instruction

- Integer Registers



1: Regularly Designed ISA

- opcode, funct: define operation
- rs1, rs2: **read** registers, rd: **write** registers
- imm (immediate): stores specified numbers



2: Reduced Instructions

- Consider only basic integer computation, (un)conditional jump, and load/store
  - RISC-V: 37 instructions (winner), MIPS R2000: 51 instructions
- Pseudo instruction

Pseudo Instruction	MIPS	RISC-V
nop	sll, $0, $0, 0	addi x0, x0, 0
not rd, rs	nor rd, rs, $0	xori rd, rs, -1
neg rd, rs	sub rd, $0, rs	sub rd, x0, rs
j offset	not pseudoinstruction	jal x0, offset
jal offset	not pseudoinstruction	jal x1, offset
jr rs	not pseudoinstruction	jalr x0, rs, 0
jalr rs	jalr $31, rs	jalr x1, rs, 0

Comparison of Instructions

MIPS

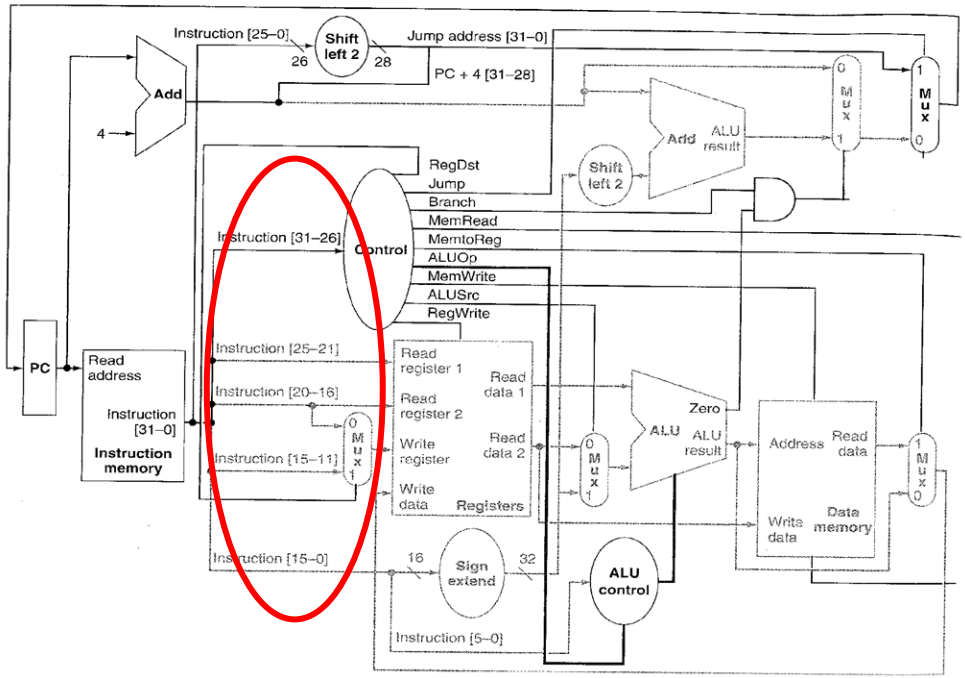
Instr	Type	Discription
nor rd, rs, rt	R	$rd = $rs ~| $rt
sll rd, rt, shamt	R	$rd = $rt << shamt
sra rd, rt, shamt	R	$rd = $rt >>> shamt
srl rd, rt, shamt	R	$rd = $rt >> shamt
j target	J	Jump to {(PC+4)[31:28],target[25:0]<<2}
jal target	J	Jump to {(PC+4)[31:28],target[25:0]<<2} $ra = PC+4
jr rs	R	Jump to $rs
jalr rs, rd	**R**	**Jump to $rs** **$rd = PC+4**

RISC-V

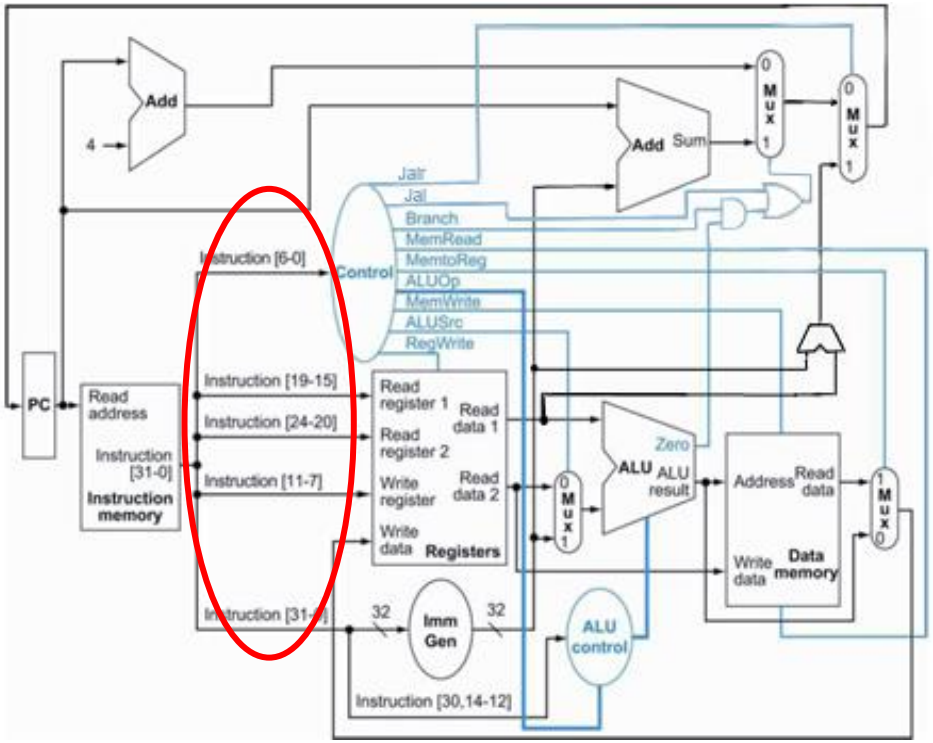
Instr	Type	Discription
Not supported		
sll rd, rs1, rs2	R	$rd = $rs1 << $rs2
sra rd, rs1, rs2	R	$rd = $rs1 >>> $rs2
srl rd, rs1, rs2	R	$rd = $rs1 >> $rs2
Pseudo Instruction		Use jal r0, imm[19:0]
jal rd, imm[19:0]	UJ	Jump to PC+J_imm[31:0], $rd = PC+4
Pseudo Instruction		Use jalr r0, 0{rs}
jalr rd, imm{rs}	**I**	**Jump to ($rs+I_imm[11:0]),** **$rd = PC+4**

TODO1: Decode Ports

MIPS

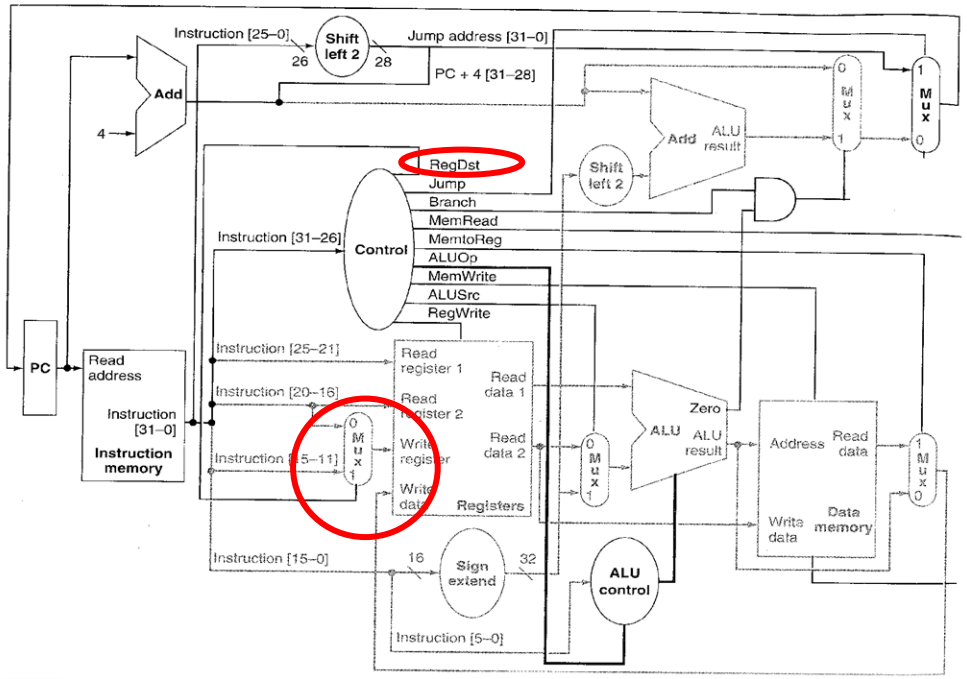


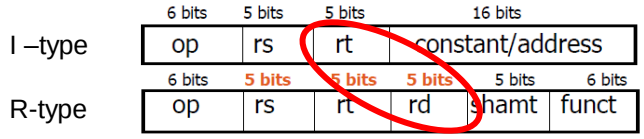
RISC-V



TODO2: Remove Mux

MIPS

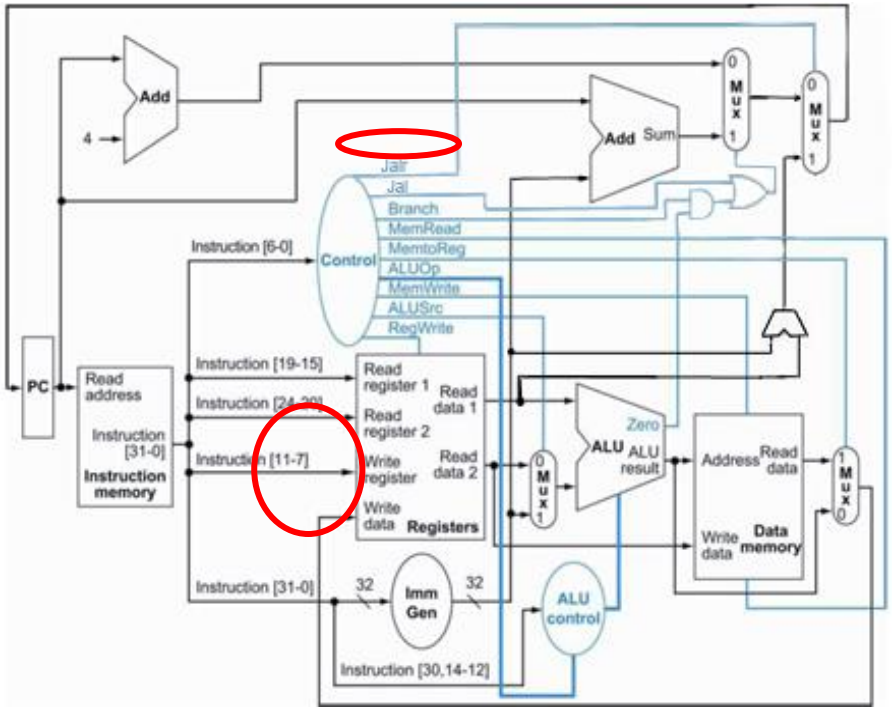


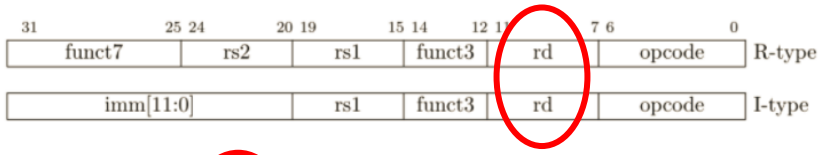


Add $rd, $rs, $rt
Ld $rt, $rs, imm[15:0]

Read reg1
Read reg2
Write reg

RISC-V





Add $rd, $rs1, $rs2
Lw $rd, $rs1, imm[11:0]

Before We Move On…

- Notice that immediate values among instructions have different meanings
- Beq
  - Immediate is encoded in ***halfword*** offset
  - Branch to an ***relative*** address (PC + Immediate)
- Jal
  - Immediate is encoded in ***halfword*** offset
  - Jump to an ***relative*** address (PC + immediate)
  - Store PC+4 to an indicated register
- Jalr
  - Immediate is encoded in ***byte*** offset
  - Jump to an ***absolute*** address (rs + immediate)
  - Store PC+4 to an indicated register

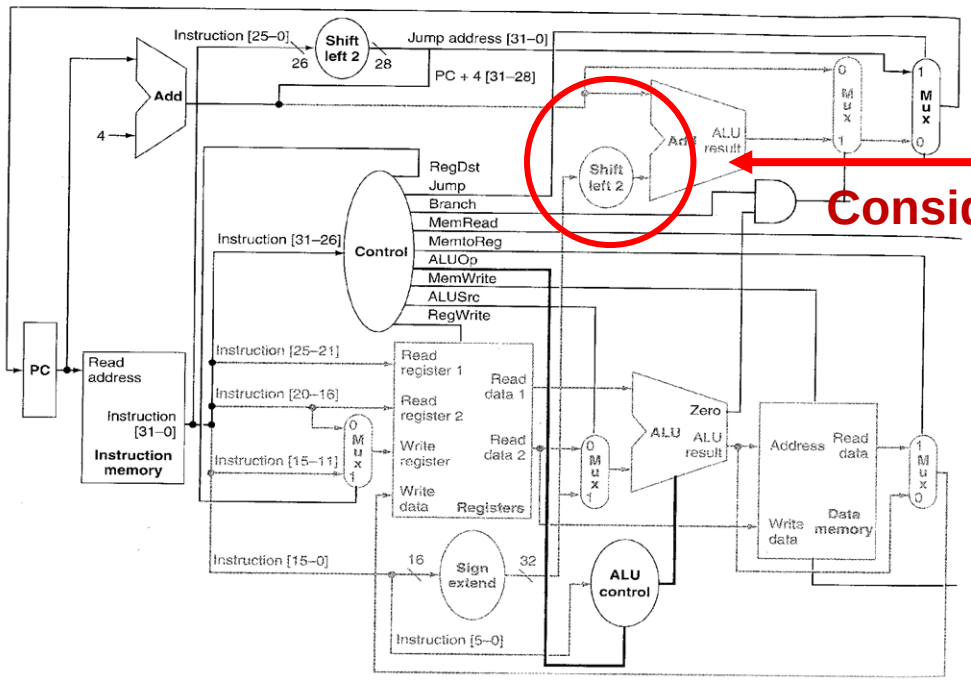
Addr	Instr
0x00	Addi x0, x0, 0
0x04	Addi x0, x0, 0
0x08	Beq x0, x0, **0x08**
0x0C	…
0x10	…
0x14	…
0x18	Jal x2, **0x04**
0x1C	…
0x20	Jalr x3, x2, **0xFF0**

Register File		
	x0	Constant 0
	x2	**0x1C**
	x3	0x24

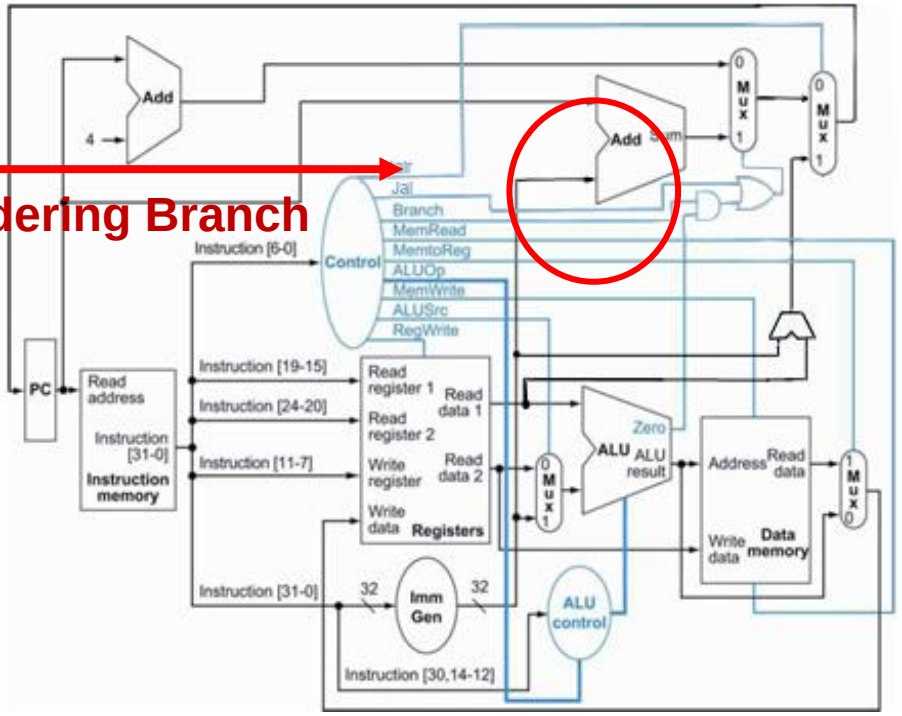
※ Due to 16 bit Compressed Instruction, half-word is used

TODO3: Remove the Shifter

MIPS



RISC-V



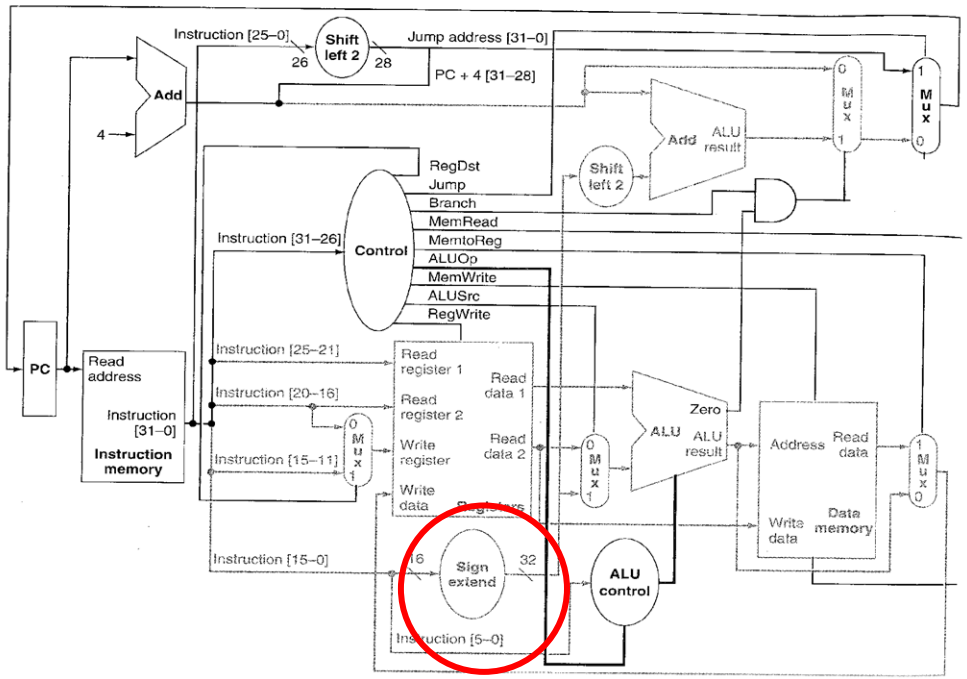
Considering Branch

- ***Immediate*** produced by Sign-extended module is in ***word offset***
- **PC + 4 + (SignExtendOut) << 2**

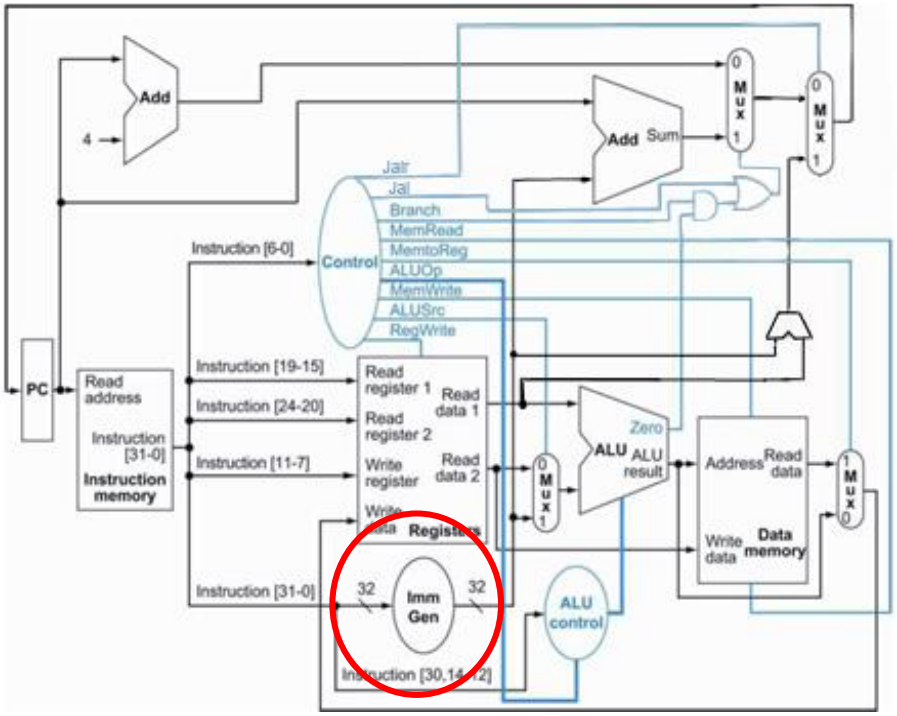
- Output of ImmGen produces 32bit ***immediate*** with ***byte offset***
- **PC + ImmGenOut (Shifted Left 1 implicitly)**

TODO4: ImmGen

MIPS



RISC-V



Immed in Word/Byte offset → **SignExtend** → Immed in Word/Byte offset

Immed in Halfword/Byte offset → **ImmGen** → Byte offset

Using Immediate Encoding Variant

TODO4: ImmGen(Cont)

❖ Immediate Encoding Varient

31	30	25	24	21	20	19	15	14	12	11	8	7	6	0	
funct7			rs2			rs1		funct3		rd			opcode		R-type
imm[11:0]						rs1		funct3		rd			opcode		I-type
imm[11:5]			rs2			rs1		funct3		imm[4:0]			opcode		S-type
imm[12]	imm[10:5]		rs2			rs1		funct3		imm[4:1]	imm[11]		opcode		B-type
imm[31:12]										rd			opcode		U-type
imm[20]	imm[10:1]			imm[11]		imm[19:12]				rd			opcode		J-type

add, sub…

lw, jalr, addi…

sw, sd…

beq

lui

jal

31	30	20	19	12	11	10	5	4	1	0	
— inst[31] —						inst[30:25]		inst[24:21]		inst[20]	I-immediate
— inst[31] —						inst[30:25]		inst[11:8]		inst[7]	S-immediate
— inst[31] —				inst[7]		inst[30:25]		inst[11:8]		0	B-immediate
inst[31]	inst[30:20]		inst[19:12]		— 0 —						U-immediate
— inst[31] —			inst[19:12]	inst[20]		inst[30:25]		inst[24:21]		0	J-immediate

Immediate encoding implicitly turns half-word offset to byte offset

ImmGen: Case Study

- beq x8 x10 0x01A //branch to PC+(26<<1)
- Immediate = $+26_{(dec)} = 0000\ 0001\ 1010_{(bin)}$ (halfword offset)

↑Imm[12] ↑Imm[1], since Imm[0] is always 0

[31]	[30:25]	[24:20]	[19:15]	[14:12]	[11:8]	[7]	[6:0]	B-type instruction
0	000001	01010	01000	000	1010	0	1100011	
Imm[12]	Imm[10:5]	rs2	rs1	funct3	Imm[4:1]	Imm[11]	opcode	

[31:12]	[11]	[10:5]	[4:1]	[0]	B-Immediate
{20{0}}	0	000001	1010	0	
–Inst[31]–	Inst[7]	Inst[30:25]	Inst[11:8]	Constant 0	

- Range of byte for conditional branch
  - 12bit immediate(half word encoded)
  - ± 2048 halfwords → ± 4096 bytes

ImmGen: Case Study(cont)

- jal x0 0x00006 //jump to PC+(6<<1)
- Immediate = $+6_{(dec)} = 0000\ 0000\ 0000\ 0000\ 0110_{(bin)}$ (halfword offset)

↑Imm[1], since Imm[0] is always 0

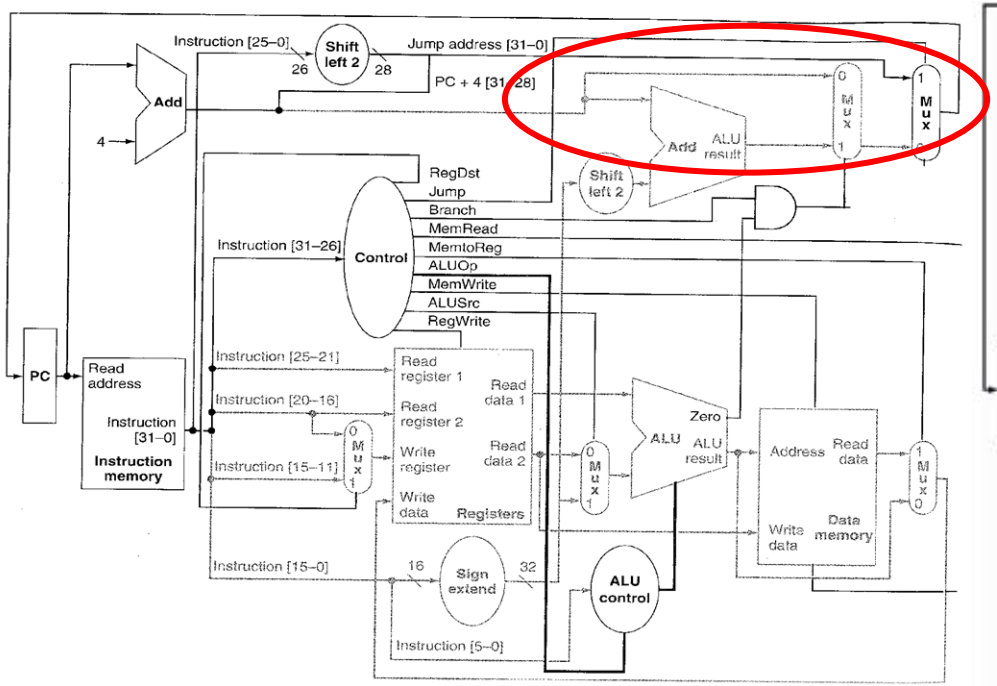
[31]	[30:21]	[20]	[19:12]	[11:7]	[6:0]	J-type instruction
0	0000000110	0	00000000	00000	1101111	
Imm[20]	Imm[10:1]	Imm[11]	Imm[19:12]	rd	opcode	

[31:20]	[19:12]	[11]	[10:5]	[4:1]	[0]	J-immediate
{12{0}}	00000000	0	000000	0110	0	
–Inst[31]–	Inst[19:12]	Inst[20]	Inst[30:25]	Inst[24:21]	Constant 0	

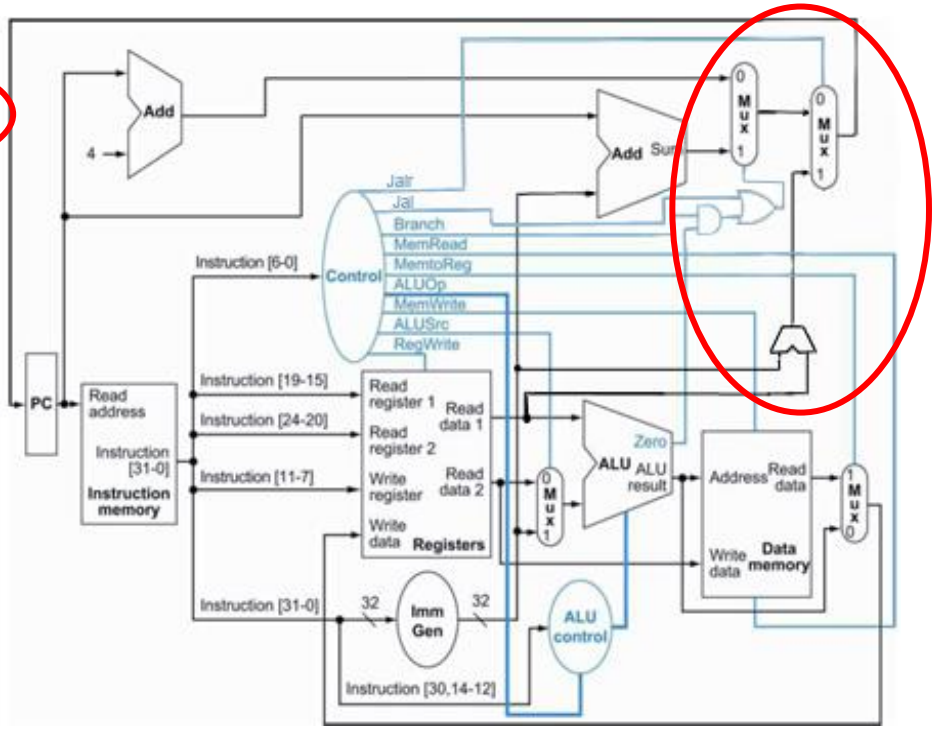
- Range of byte for unconditional jump
  - 20bit immediate(half word encoded)
  - $\pm 2^{19}$ halfwords → ± 1 MiB

TODO5: PC, Jal, Jalr

MIPS



RISC-V



PC = (Branch) ? BranchAddr :
(Jump) ? JumpAddr :
PC+4

PC = (Branch | Jal) ? B/Jal Addr :
(Jalr) ? JalrAddr :
PC+4

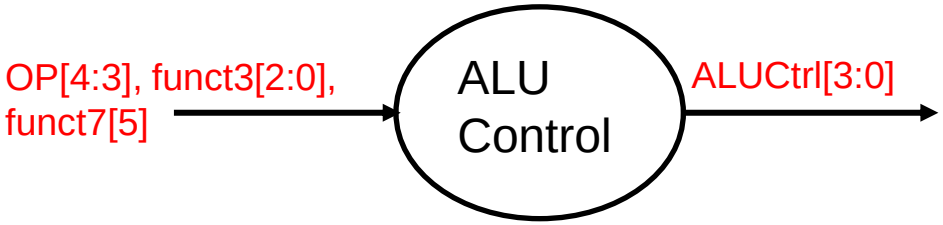
TODO6: ALU Control Signal

- For HW3, it is possible to take only funct7[5], funct3 and OP[4:3] into consideration

funct7			funct3		OP	
IR[31:25]	IR[24:20]	IR[19:15]	IR[14:12]	IR[11:7]	IR[6:0]	Inst
0000000	Rs2	Rs1	000	Rd	0110011	ADD
0100000	Rs2	Rs1	000	Rd	0110011	SUB
0000000	Rs2	Rs1	111	Rd	0110011	AND
0000000	Rs2	Rs1	110	Rd	0110011	OR
0000000	Rs2	Rs1	010	Rd	0110011	SLT
Imm[11:5]	Rs2	Rs1	010	Imm[4:0]	0100011	SW
Imm[11:0]		Rs1	010	Rd	0000011	LW
Imm[12|10:5]	Rs2	Rs1	000	Imm[4:1|11]	1100011	BEQ
Imm[20|10:1|11|19:12]				Rd	1101111	JAL
Imm[11:0]		Rs1	000	Rd	1100111	JALR

TODO6: ALU Control Signal(Cont)

ALU Control Lines	Functions	Instructions
0000	AND	AND
0001	OR	OR
0010	ADD	SW, LW
0110	SUBSTRACT	SUB, Branch
1000	SLT	SLT



❖ **One of many solutions for ALUCtrl :**

ALUCtrl[0] = OP[4] & funct3[2] & funct3[1] & (!funct3[0])
ALUCtrl[1] = !(OP[4] & (!OP[3]) & funct3[1])
ALUCtrl[2] = ((!OP[4]) & (!funct3[1])) | (funct7[5] & OP[4])
ALUCtrl[3] = OP[4] & (!funct3[2]) & funct3[1]

Before We Move On: Endian

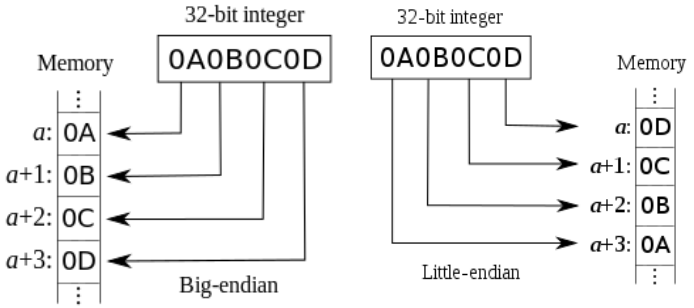
MIPS (Big-endian)

- add $r1, $r2, r3 # $r1 = $r2 + $r3

↓31bit ↓0bit

- 000000 00010 00011 00001 00000 000000
  OP $rs = $r2 $rt = $r3 $rd = $r1 shamt=0 funct = 0
- 00 43 08 00 (Instruction in Big Endian & Hex)
  [31:16] [15:0]

128x32 SRAM	[31:24]	[23:16]	[15:8]	[7:0]
0	00	43	08	00
:				
127				



RISC-V (Little-endian)

- add $r1, $r2, $r3 # $r1 = $r2 + $r3

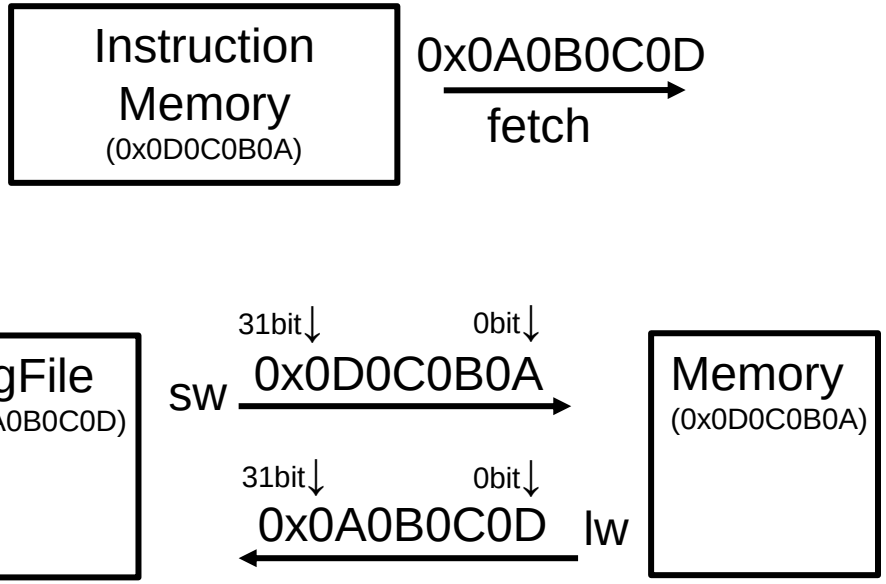
↓31bit ↓0bit

- 0000000 00011 00010 000 00001 0110011
  funct7 $rs2=$r3 $rs1=$r2 funct3 $rd=$r1 OP
- Instructions are stored in memory in a little-endian sequence of **bytes**
- 00 31 00 B3 (hex) → Little Endian → B3 00 31 00(hex)
  [31:16] [15:0] [15:0] [31:16]

128x32 SRAM	[31:24]	[23:16]	[15:8]	[7:0]
0	B3	00	31	00
:				
127				

TODO7: Data Format

- Be cautious with endian when load/store data
  - Need to perform conversion when encountering the interface of SRAM
  - Assume data 0x0A0B0C0D



Appendix A

- Why Little endian?
  - Fetch with the same address if a given value is stored in different width
    - 32bit 0x0D0C0B0A
    - 64bit 0x000000000D0C0B0A
    - We can always fetch the lowest 32bit address
  - Mainstream
    - Intel x86