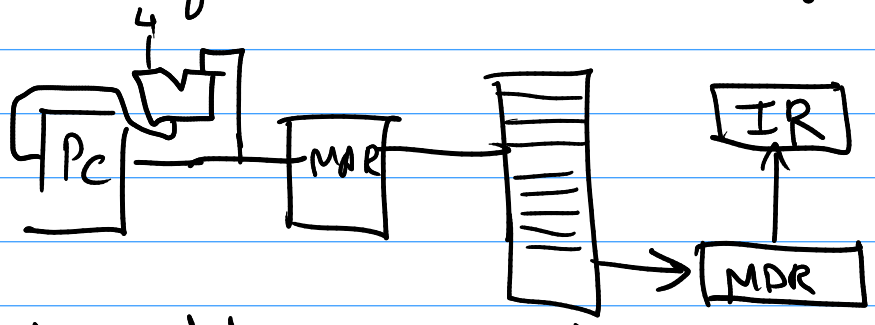
Processor design

fetch

PC next instruction → done in meantime during exec

- fetch instrn & inc PC [instrn in IR]
- use fields of instructions to select registers to read



- execute depending on instruction

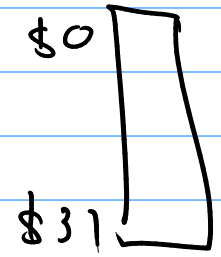
→ repeat

Why instruction mem & data mem

Ans: (4 bytes) (different sizes)

Either physical or logical division

Register file:



Single cycle vs multicycle vs pipelined

Centire instruction in 1 clock cycle
time as fast as slowest instruction

each step of instruction in 1 clock cycle
instruction takes how much ever it needs (dynamic time/cycles)

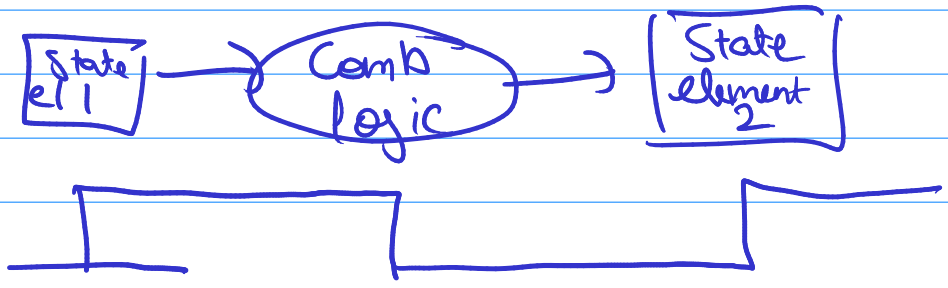
process multiple instructions as in assembly line
diff stages of (Fetch-Decode-Execute) at same time

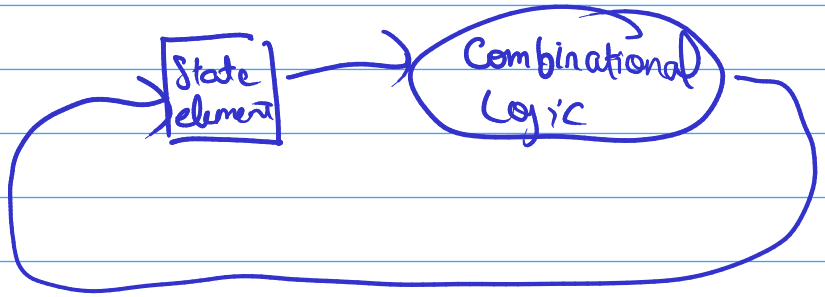
Next lecture

Functional Elements first

Combinational Elements (depends only on current values) Operate on data
Sequential Elements (State) (Contain data)

Eg: Comb.: ALU

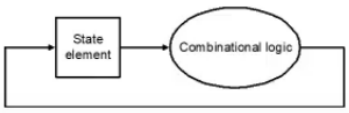




Combinational Elements

- Works as an *input* ⇒ *output function*, e.g., ALU
- Combinational logic *reads input data from one register and writes output data to another, or same, register*
  - read/write happens in a single cycle – combinational element cannot store data from one cycle to a future one

Combinational logic hardware units



Sequential / State Elements

define state of a machine

FF/Latch 1-bit state/memories elts

Sequential or State Elements

- State elements contain *data* in internal storage, e.g., *registers* and *memory*
- All state elements together *define* the *state of the machine*
  - *What does this mean? Think of shutting down and starting up again…*
- *Flipflops* and *latches* are 1-bit state elements, equivalently, they are *1-bit memories*
- The *output*(s) of a flipflop or latch *always* depends on the bit value stored, i.e., its state, and can be called *1/0* or *high/low* or *true/false*
- The *input* to a flipflop or latch can change its state depending on whether it is clocked or not…

State element

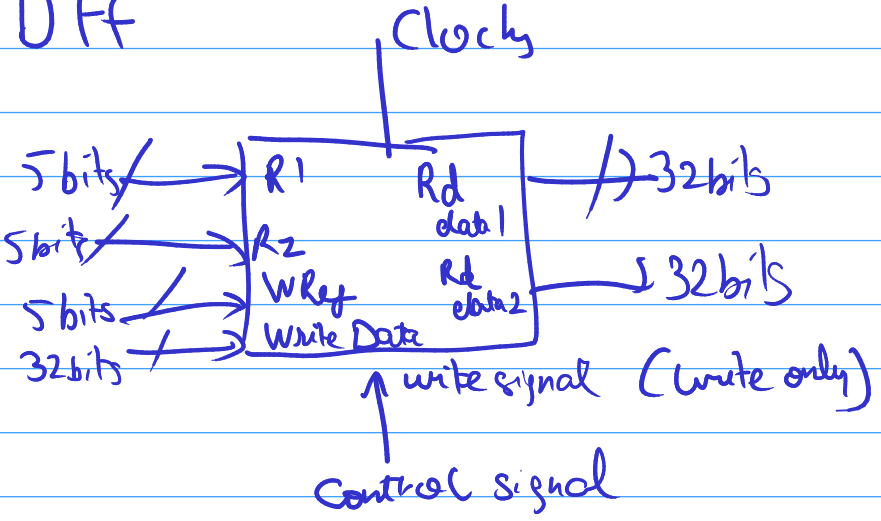
Register file

Arrays of DFF

R-type

$0 – $31

@posedge clock

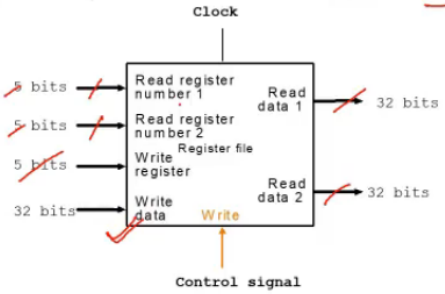


State elements on the Datapath: Register File

- Registers are implemented with arrays of D-flipflops

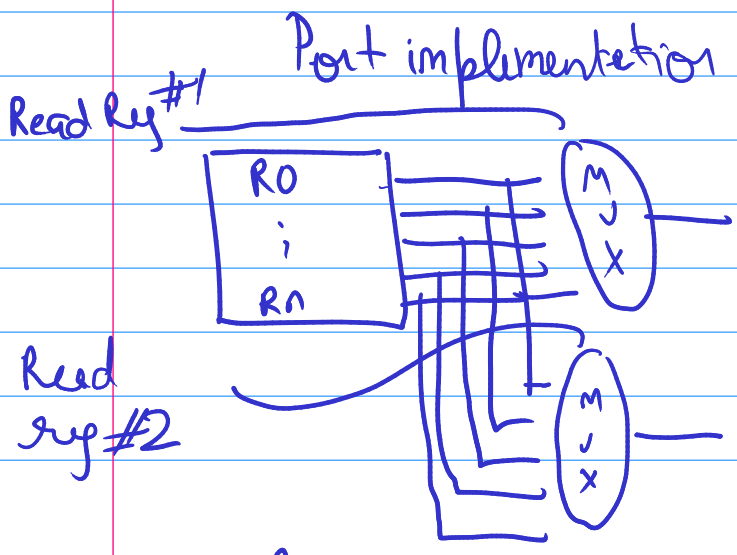
$0 – $31

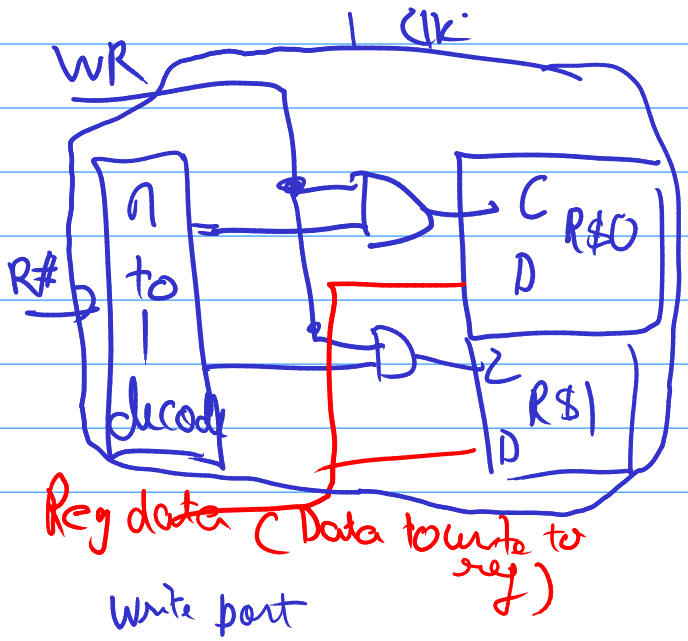
R-type



Register file with two read ports and one write port

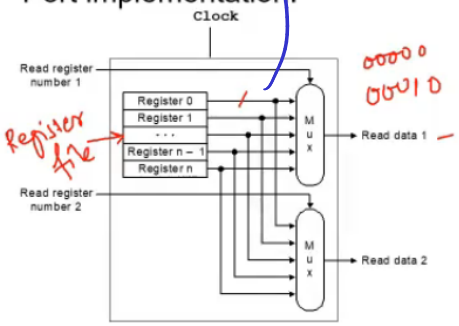
Port implementation



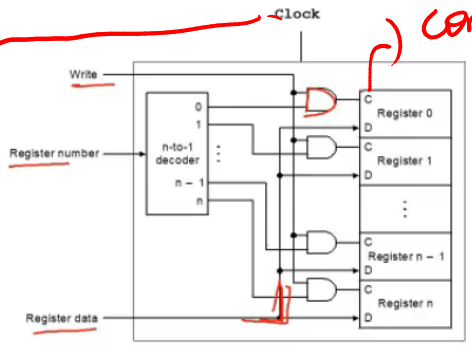


State elements on the Datapath: Register File

- Port implementation:



Read ports are implemented with a pair of multiplexors – 5 bit multiplexors for 32 registers



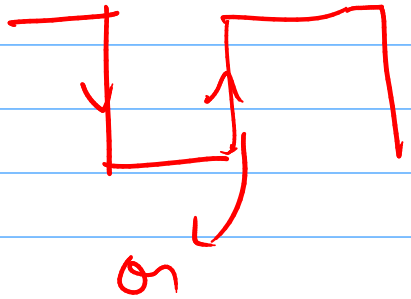
Write port is implemented using a decoder – 5-to-32 decoder for 32 registers. Clock is relevant to write as register state may change only at clock edge

Clock connected to all regs

Single Cycle

Single long cycle

either



or

Single Cycle Implementation

- Our first implementation of MIPS will use a *single* long clock cycle for every instruction
- Every instruction begins on one up (or down) clock edge and ends on the next up (or, down) clock edge
- This approach is *not practical* as it is much slower than a *multicycle* implementation where different instruction classes can take different numbers of cycles
  - in a single-cycle implementation every instruction must take the same amount of time as the slowest instruction
  - in a multicycle implementation this problem is avoided by allowing quicker instructions to use fewer cycles
- Even though the single-cycle approach is not practical it is simple and useful to understand first.

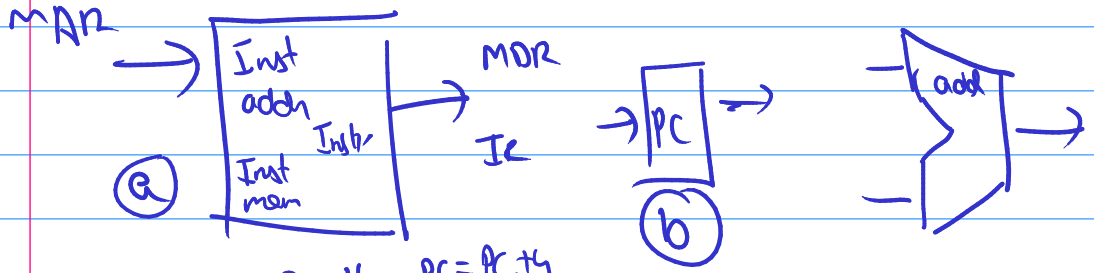
Not practical → too slow (you need to keep freq as slowest instruction)

General Sequence of Instruction Execution

- For every instruction:
  - Send the *program counter* (PC) to the memory that contains the code and fetch the instruction from that memory.
  - Read one or two registers, using fields of the instruction to select the registers to read.
- For the load/store word instruction, we need to read only one register.
- Three types of instructions: memory-reference, arithmetic-logical, and branches. beq, j lw/sw add slt
- All instruction classes, except jump, use the ALU after reading the registers.

as j -> just change PC others need to inc PC -> PC+4

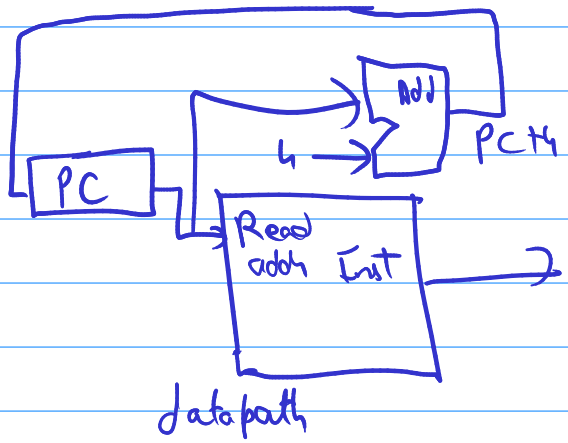
Datapath Store/Fetch inst & PC inst



MAR ← PC then PC = PC+4

MDR ← Mem

IR



Store & fetch inc PC

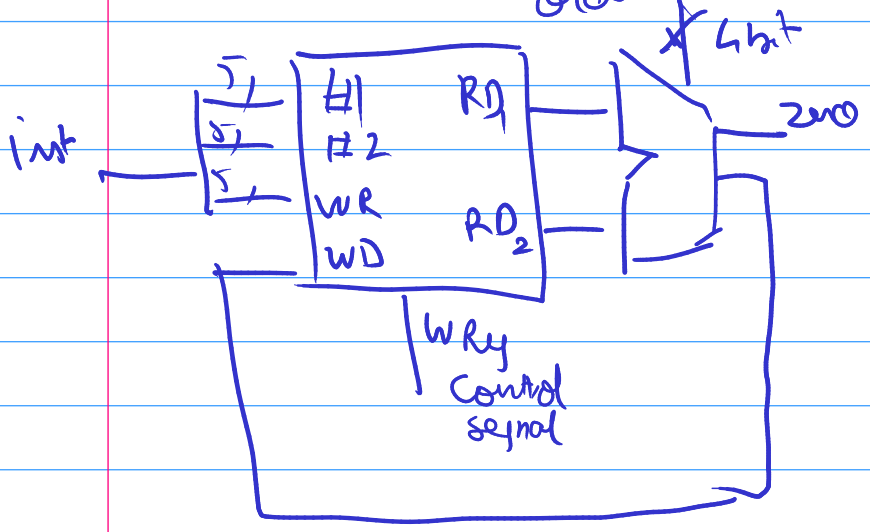
Inst ← MEM[PC]
PC ← PC+4

[micro instructions]

R type

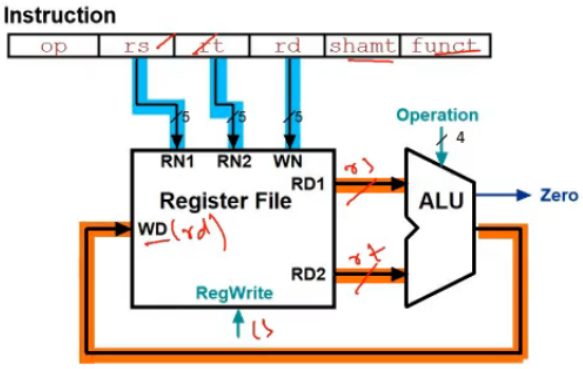
opc src target dst shamt func

machine code order



Check this for assembly code order

Datapath - Visualizing




```
add rd, rs, rt
```

R[rd] <- R[rs] + R[rt];

Load store

2 regs, source offset

base addr 16bit offset

-ve +ve

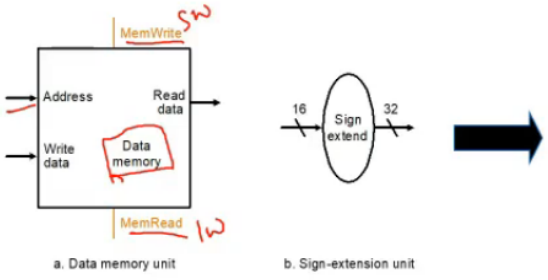
16 bit → Sign extend → 32 bit

MSB 0 all zero

MSB 1 all significant bits 1

Data memory

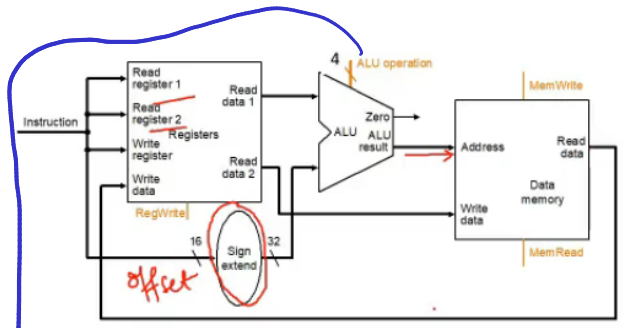
Datapath – Load/Store Instruction



a. Data memory unit

b. Sign-extension unit

Two additional elements used to implement load/stores



Datapath

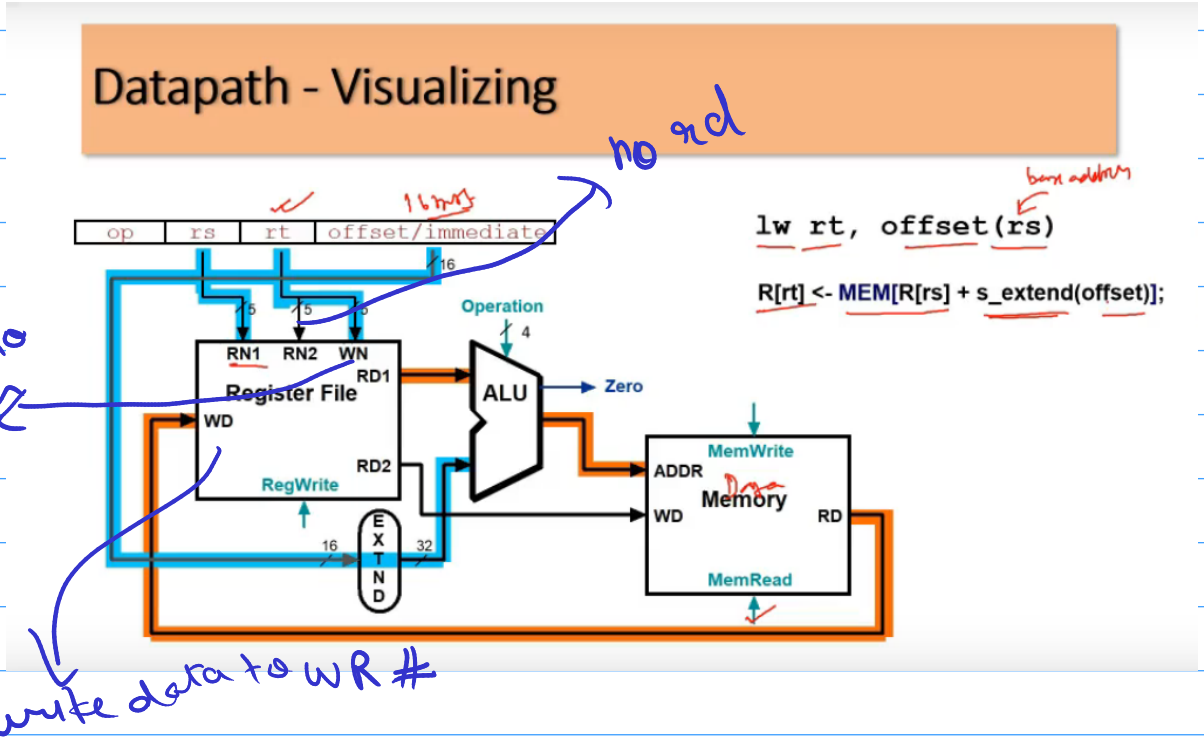
This is not opcode

bcuz

lw rt, offset(rs)

r(rt) ← Mem[R[rs] + s_extend(offset)];

op rs rt offset/imm16



no wrd

write to register

write data to WR #

sw rt, offset(rs)

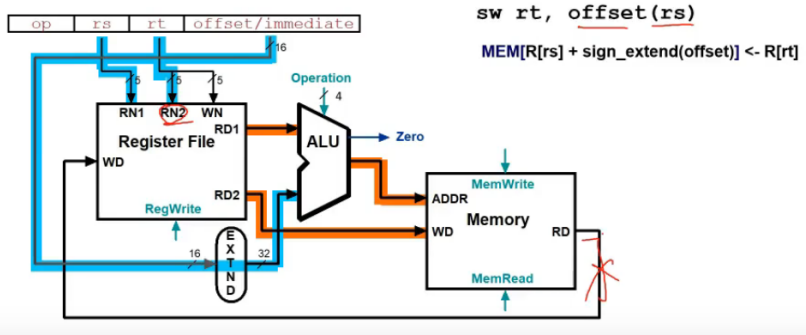
Mem[R[rs] + signext(offset)] ← R[rt]

RN2 is ~~not~~ used,

Write register (WN) isn't

RD doesn't happen

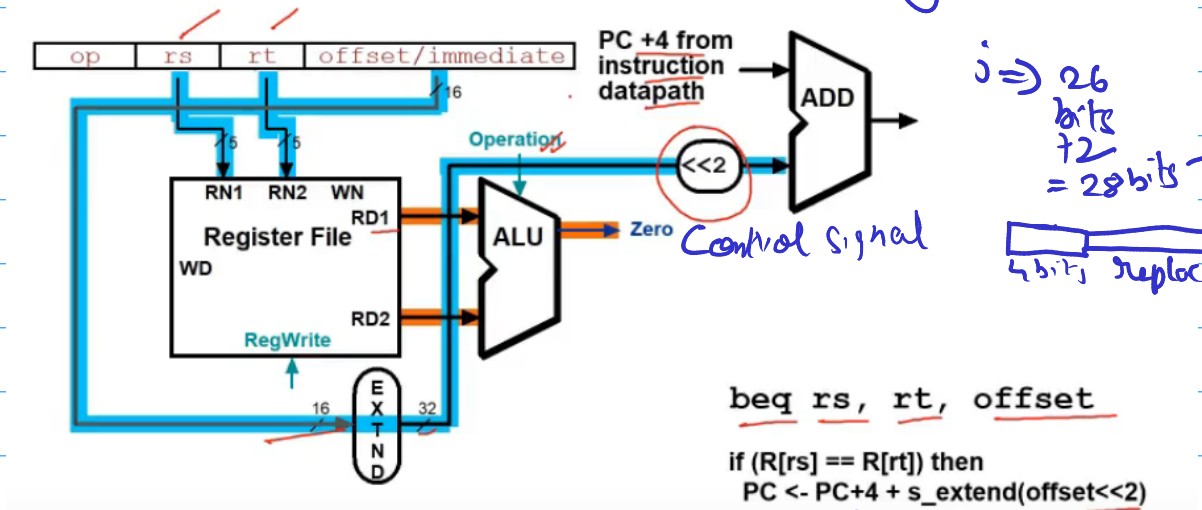
WD does



Branch instn

No shift hw reqd

simply connect wires ~~act~~ each shifted by 2 bits



i ⇒ 26 bits ↑2 = 28 bits

4 bits replace

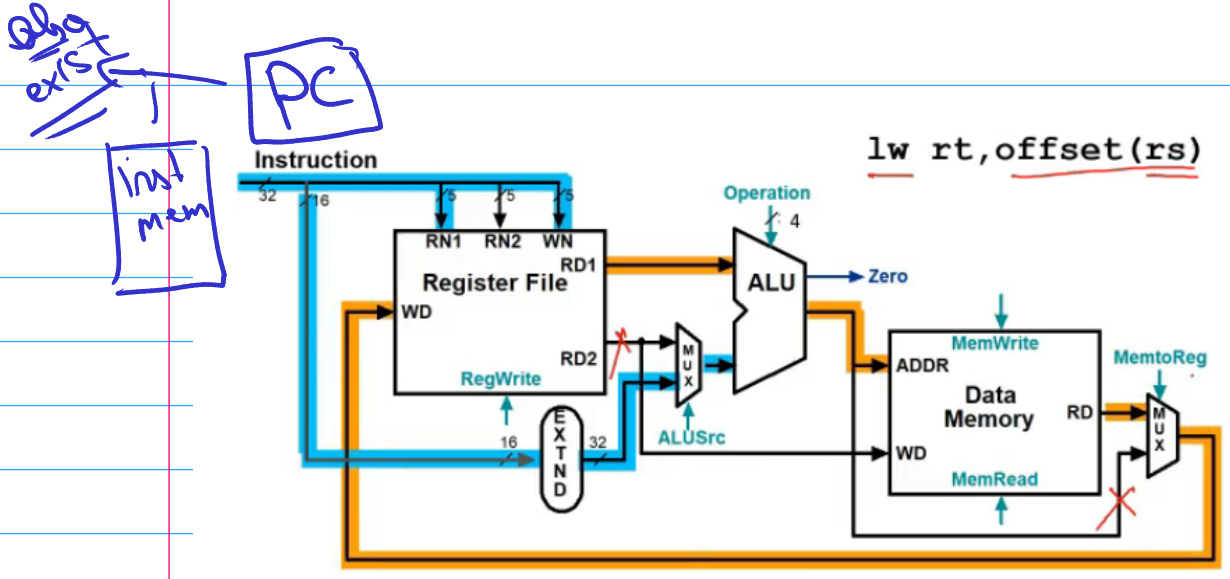
for branch you need to shift

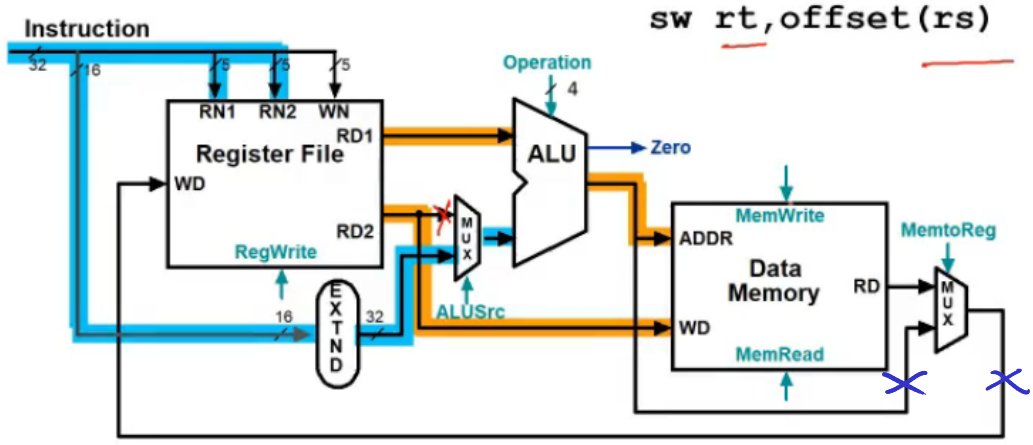
Concatenation CMIPS auch how

4bits of PC + 26 bits + 00

32 bits unconditional jump

Datapath without Branch





Why are we using different instruction mem & data memory

In a single cycle we need to

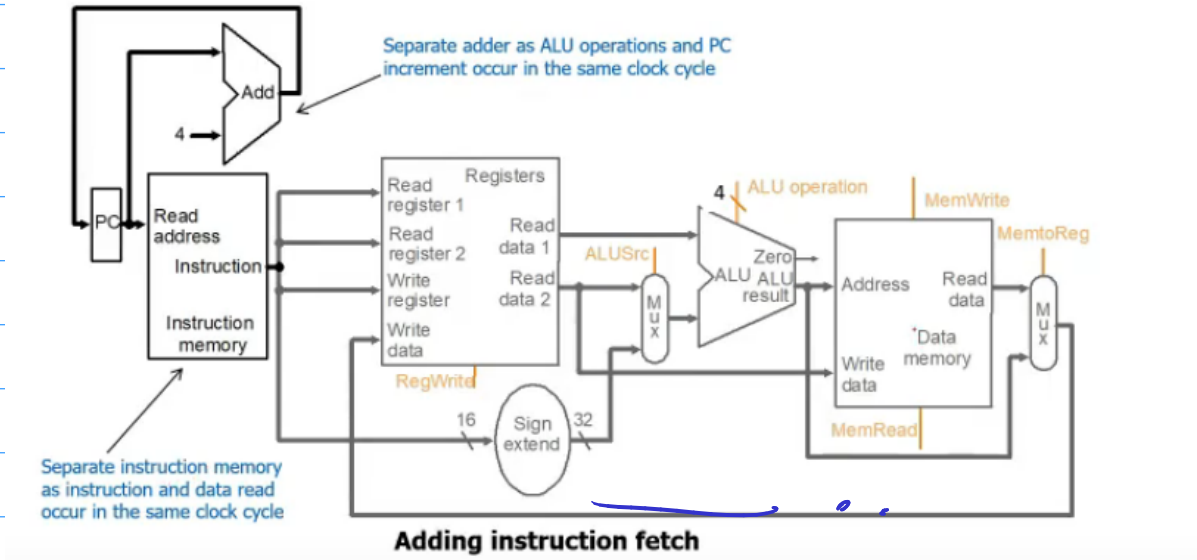
(i) read from memory (instruction)

(ii) write/read to memory (data)

So the necessary signals must be generated, all in 1 cycle

⇒ not possible (well technically yeah but like, too complex ig)

We either both read & write (sw)
or both read & read (lw)
but in 1 cycle x D



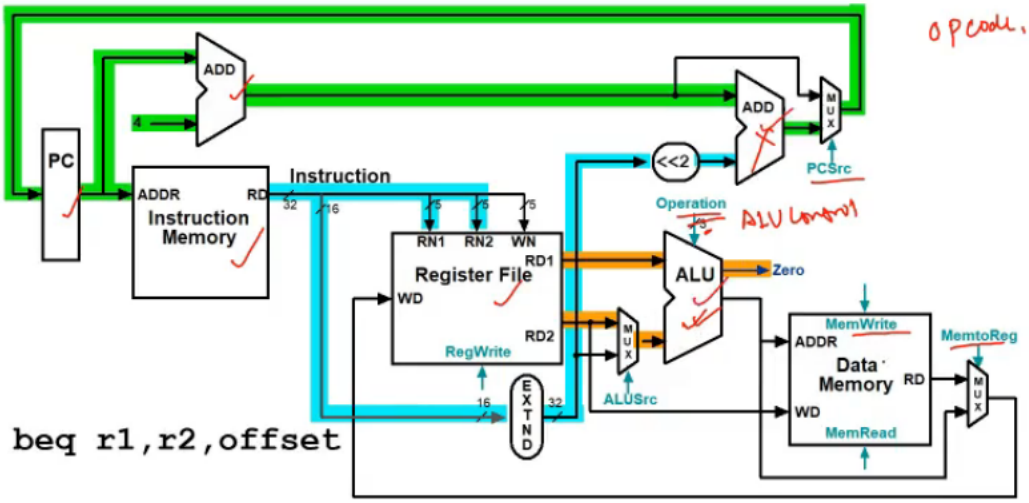
Adding instruction fetch

R' type or I type

Lecture end (OMFG)

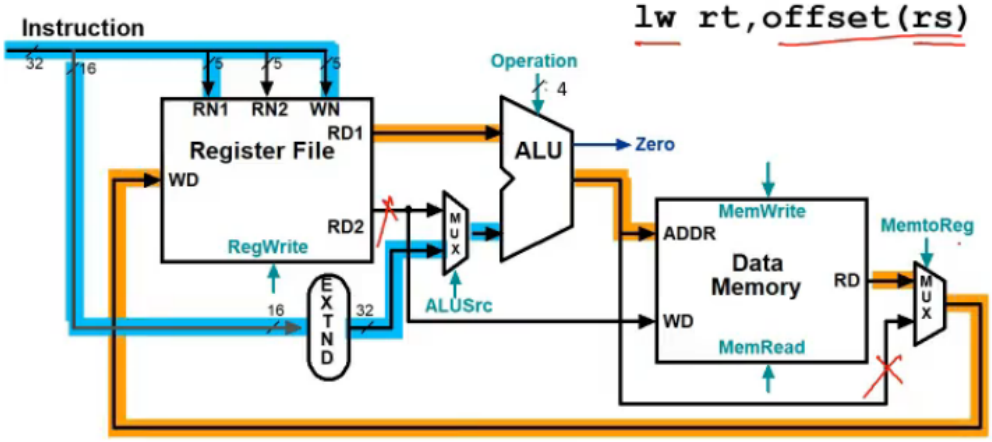
Single cycle not ended ((

opcode, function

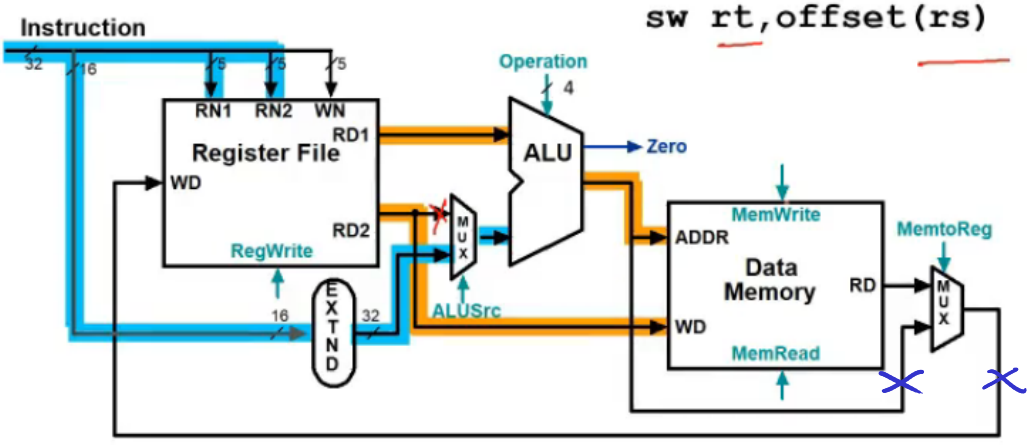


beq r1,r2,offset

lw rt,offset(rs)

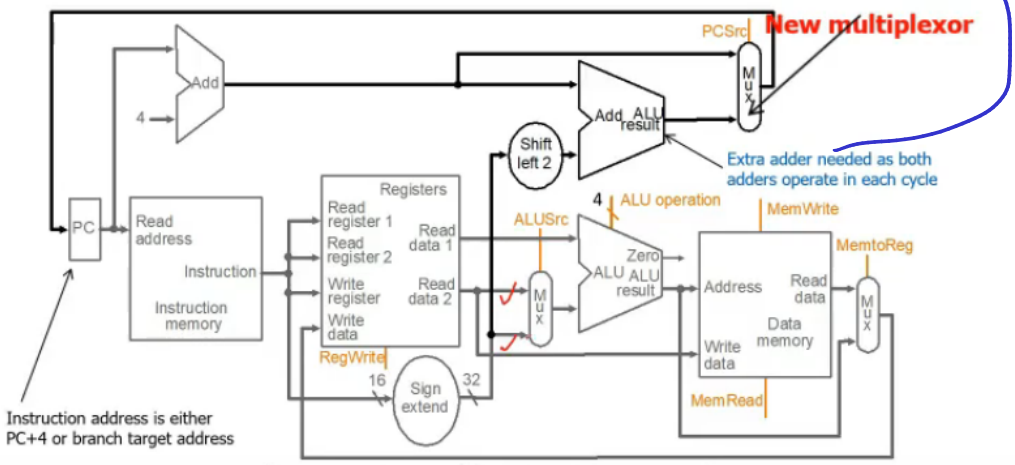


sw rt,offset(rs)



Is there an operation where we use both ALUs??

I don't quite get this part



Adding branch capability and another multiplexor

Control Unit

Who sets ?

- Control unit takes input from
  - the instruction opcode bits
- Control unit generates
  - ALU control input
  - write enable (possibly, read enable also) signals for each storage element
  - selector controls for each multiplexor

register file doesn't need (register read) not used

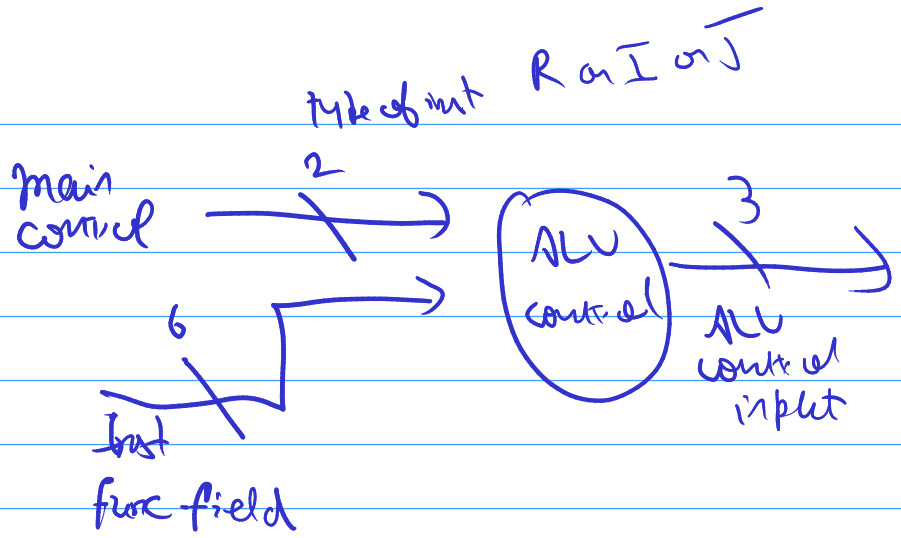
memory read

register memory (Data memory)

ALU → R type → (add, sub), slt

I type → (lw, sw), beq, pc+4+offset → separate add logic (the top adder)

X J → j



Setting ALU Control Bits

Instruction opcode	AluOp	Instruction operation	Funct Field	Desired ALU action	ALU control input
LW	00	load word	xxxxxx	add	010
SW	00	store word	xxxxxx	add	010
Branch eq	01	branch eq	xxxxxx	subtract	110
R-type	10	add	100000	add	010
R-type	10	subtract	100010	subtract	110
R-type	10	AND	100100	and	000
R-type	10	OR	100101	or	001
R-type	10	set on less	101010	set on less	111

to decide we do 3th else (bar)

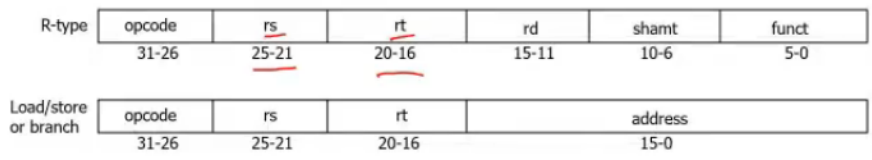
either lw or sw

ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	010
0	1	X	X	X	X	X	X	110
1	X	X	X	0	0	0	0	010
1	X	X	X	0	0	1	0	110
1	X	X	X	0	1	0	0	000
1	X	X	X	0	1	0	1	001
1	X	X	X	1	0	1	0	111

rs is always same (pos is same) bits

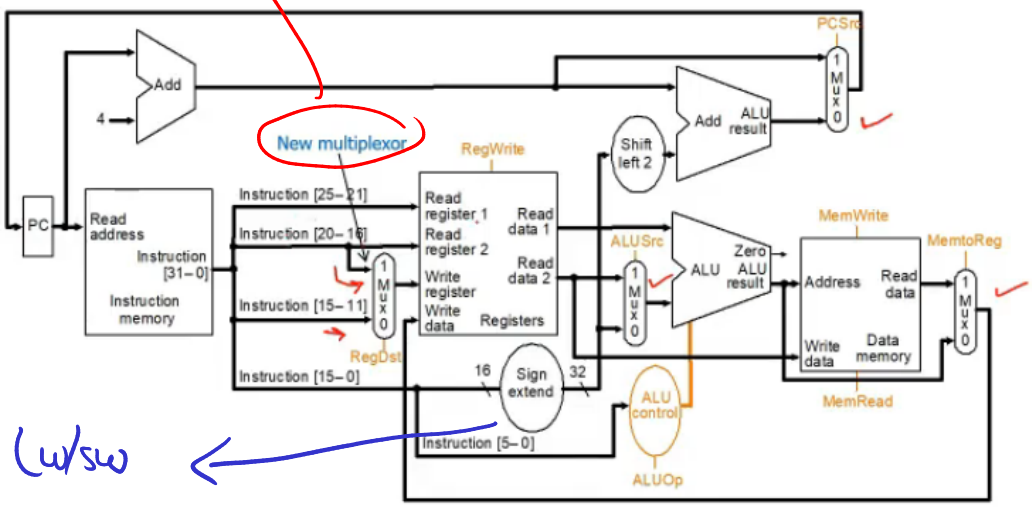
I type has diff opcodes

R type has diff function codes (opcode is '0')



- Observations about MIPS instruction format
  1. opcode is always in bits 31-26
  2. two registers to be read are always rs (bits 25-21) and rt (bits 20-16)
  3. base register for load/stores is always rs (bits 25-21)
  4. 16-bit offset for branch equal and load/store is always bits 15-0
  5. destination register for loads is in bits 20-16 (rt) while for R-type instructions it is in bits 15-11 (rd) (*will require multiplexor to select*)

for selecting reg to write to (either rt or rd)



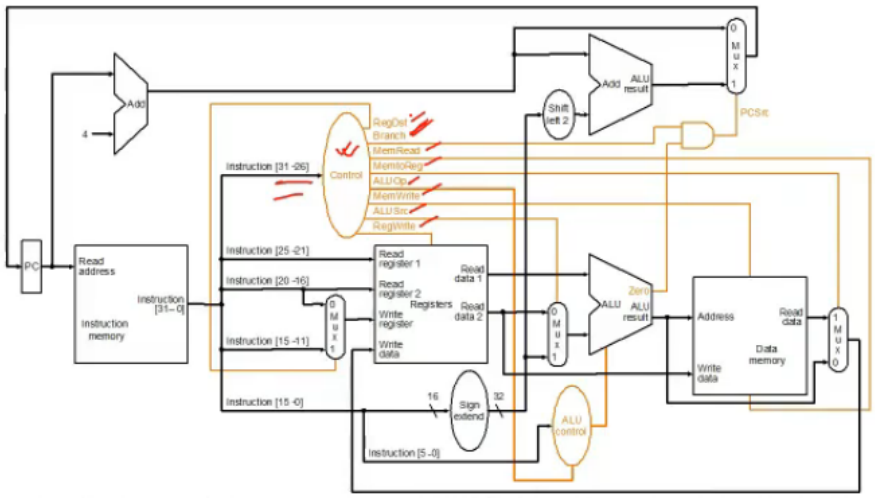
Adding control to the MIPS Datapath III (and a new multiplexor to select field to specify destination register): what are the functions of the 9 control signals?

Control Signals

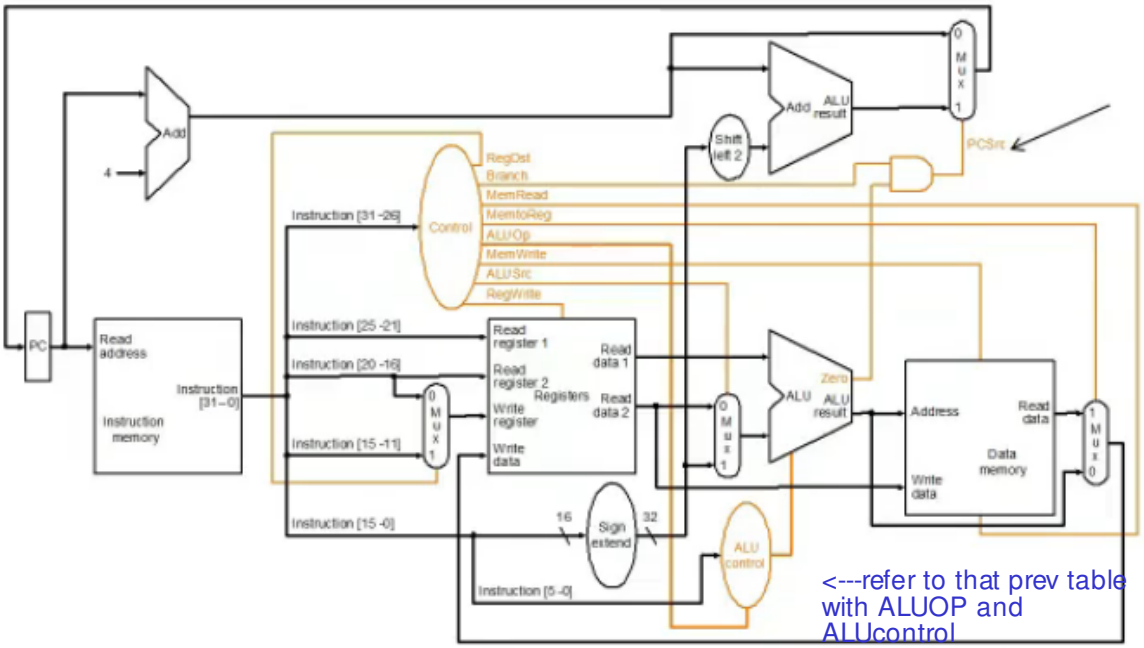
Signal name	Effect when deasserted	Effect when asserted
RegDst	The register destination number for the Write register comes from the rt field (bits 20-16)	The register destination number for the Write register comes from the rd field (bits 15-11)
RegWrite	None	The register on the Write register input is written with the value on the Write data input
AlUSrc	The second ALU operand comes from the second register file output (Read data 2)	The second ALU operand is the sign-extended, lower 16 bits of the instruction
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4	The PC is replaced by the output of the adder that computes the branch target
MemRead	None	Data memory contents designated by the address input are put on the first Read data output
MemWrite	None	Data memory contents designated by the address input are replaced by the value of the Write data input
MemtoReg	The value fed to the register Write data input comes from the ALU (sw)	The value fed to the register Write data input comes from the data memory (lw)

Datapath with Control - II

2 new things ALU OP & branch



MIPS datapath with the control unit: input to control is the 6-bit instruction opcode field, output is seven 1-bit signals and the 2-bit ALUOp signal



Determining control signals for the MIPS datapath based on instruction opcode

Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	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

it's kinda hard to memorize all this, revise this twice. After that's it's quite logical anyways

Why do you have an ALUop and function code, why not just one ;-;?

Ans:
i) As this is multilevel decoding
-- it's easier to design control logic this way

RISC

Opcode is completely decoded in CISC
(not really having control signals)

ii) Several instructions have very similar datapaths,
instead of designing a single opcode and having a very large decoding unit we can divide them

Opcode → Control signals → rdest (not rd!) rd memread

↓

ALU operations

↳ 2 bits of ALU op + 6 bits of funct code } 3 bits of ALU control

& making the control logic is outside the scope, in appendix B

→ Clock is inside control unit mostly

Sometimes

@ each the edge control signal generated

↳ here clock is inside control unit

sometimes not

↳ here clock is outside (in some archs)

so some synchronization is required

(like essentially, it depends on implementation)