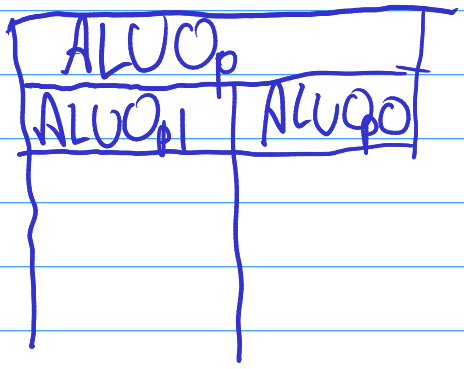
Oct 5th

Multicycle CPU Design

We didn't quite finish control unit just yet

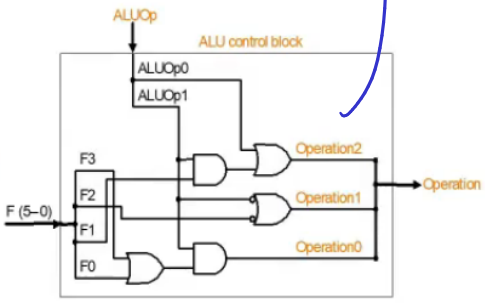


only 3 operations shown

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

Truth table for ALU control bits

lw/sw
branch/j
R-type



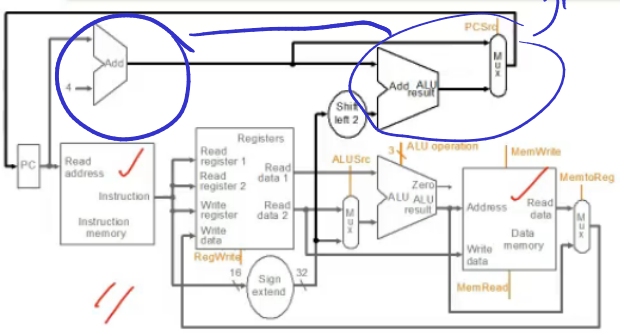
ALU control logic

- One solution: a variable-period clock with different cycle times for each instruction class
  - *unfeasible*, as implementing a variable-speed clock is technically difficult
- Another solution:
  - use a smaller cycle time…
  - …have different instructions take different numbers of cycles
  - by breaking instructions into steps and fitting each step into one cycle
  - *feasible: multicyle approach*!

Too ideal

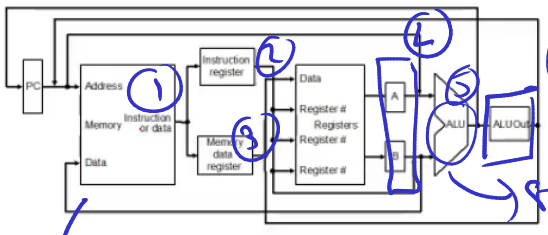
- Break up the instructions into *steps*
  - each step takes one clock cycle
  - balance the amount of work to be done in each step/cycle so that they are about equal
  - restrict each cycle to use at most once each major functional unit so that such units do not have to be replicated
  - functional units can be shared between different cycles within one instruction
- Between steps/cycles
  - At the end of one cycle store data to be used in *later cycles of the same* instruction
    - need to introduce additional *internal* (programmer-invisible) registers for this purpose
  - Data to be used in *later instructions* are stored in programmer-visible state elements: the register file, PC, memory

"Shared"



Single-cycle datapath

no longer needed



Multicycle datapath (high-level view)

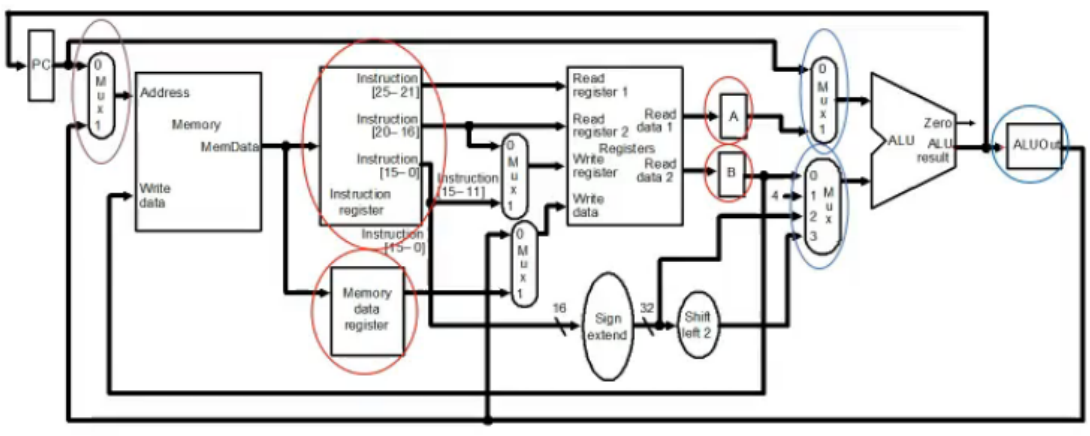
single ALU

single memory

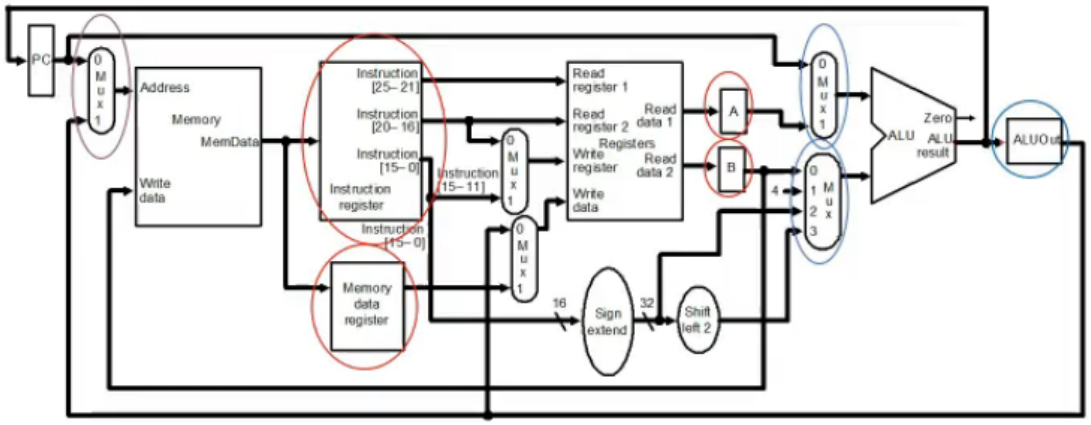
- Note particularities of multicyle vs. single cycle diagrams
  - single memory for data and instructions
  - single ALU, no extra adders
  - extra registers to hold data between clock cycles

Instruction Reg: Current inst to be exec.

MDR - data to be used



Basic multicycle MIPS datapath handles R-type instructions and load/stores: new internal register in red ovals, new multiplexors in blue ovals



Basic multicycle MIPS datapath handles R-type instructions and load/stores: new internal register in red ovals, new multiplexors in blue ovals

if A B were not there Current instruction executes immediately then the next instruction will not be able to decode

[D&E phase is 1 phase now]

$I_1 \rightarrow F \rightarrow D\&E$

$I_2 \rightarrow F$ ~~$\rightarrow D$~~ (since I_1 is in decode & execute phase)

AluOut is to be storing the result to prevent immediately writing to data

⇒ next instruction might end up overwriting the stuff

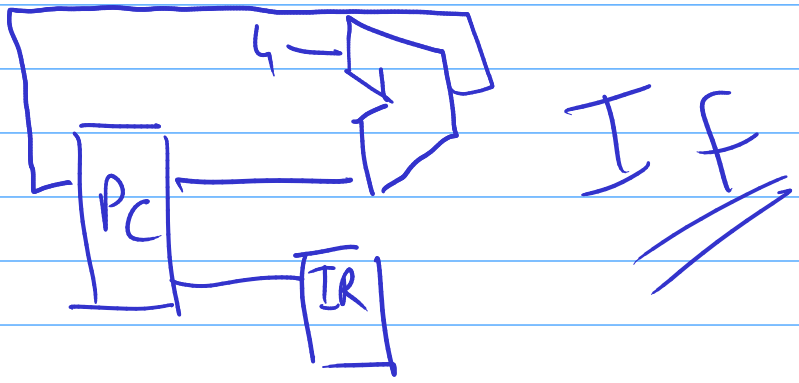
- Our goal is to break up the instructions into *steps* so that
  - ✓ each step takes one clock cycle
  - ✓ the amount of work to be done in each step/cycle is about equal
  - ✓ each cycle uses at most once each major functional unit so that such units do not have to be replicated
  - ✓ functional units can be shared between different cycles within one instruction
- Data at end of one cycle to be used in next *must be stored* !!
- *We break instructions into the following potential execution steps – not all instructions require all the steps – each step takes one clock cycle*
- *Instruction fetch and PC increment (IF)*
- *Instruction decode and register fetch (ID)*
- *Execution, memory address computation, or branch completion (EX)*
- *Memory access or R-type instruction completion (MEM)*
- *Memory read completion (WB)*
- *Each MIPS instruction takes from 3 – 5 cycles (steps)*

IF & PC Increment

- Use PC to get instruction and put it in the instruction register.
- Increment the PC by 4 and put the result back in the PC.

- Can be described using *RTL* (*Register-Transfer Language*):

```
IR = Memory[PC];
PC = PC + 4;
```



ID & Register fetch

- Read regs rs & rt in case we need them
- Compute the branch address in case the instruction is a branch

RTL: A = Reg [IR[25-21]];

B = Reg [IR[20-16]];

ALUOut = PC + (sign_extend (IR[15-0]) << 2);

↳ (for BRANCH)

Step 3

- ALU performs one of four functions *depending* on instruction type
  - memory reference:
    `ALUOut = A + sign-extend(IR[15-0]);` lw/sw
  - R-type:
    `ALUOut = A op B;`
  - branch (instruction *completes*):
    `if (A==B) PC = ALUOut;`
  - jump (instruction *completes*):
    `PC = PC[31-28] || (IR(25-0) << 2)`

base → offset

↓ concat

- Again depending on instruction type:
- Loads and stores access memory
  - load
  - `MDR = Memory[ALUOut];`
  - store (instruction completes)
  - `Memory[ALUOut] = B;`
- R-type (instructions completes)
  `Reg[IR[15-11]] = ALUOut;`

- Again *depending* on instruction type:
- Load writes back (instruction *completes*)
  `Reg[IR[20-16]]=MDR;`

- Important: There is no reason from a datapath (or control) point of view that Step 5 cannot be eliminated by performing
  `Reg[IR[20-16]]= Memory[ALUOut];`
  for loads in Step 4. This would eliminate the MDR as well.

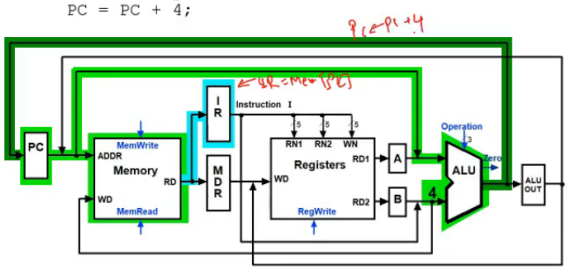
- The reason this is not done is that, to keep steps balanced in length, the design restriction is to allow each step to contain *at most* one ALU operation, or one register access, or one memory access.

Step	Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
1: IF	Instruction fetch	IR = Memory[PC] PC = PC + 4			
2: ID	Instruction decode/register fetch	A = Reg [IR[25-21]] B = Reg [IR[20-16]] ALUOut = PC + (sign-extend (IR[15-0]) << 2)			
3: EX	Execution, address computation, branch/ jump completion	ALUOut = A op B	ALUOut = A + sign-extend (IR[15-0])	if (A ==B) then PC = ALUOut	PC = PC [31-28] II (IR[25-0]<<2)
4: MEM	Memory access or R-type completion	Reg [IR[15-11]] = ALUOut	Load: MDR = Memory[ALUOut] or Store: Memory [ALUOut] = B		
5: WB	Memory read completion		Load: Reg[IR[20-16]] = MDR		

What about I type ??

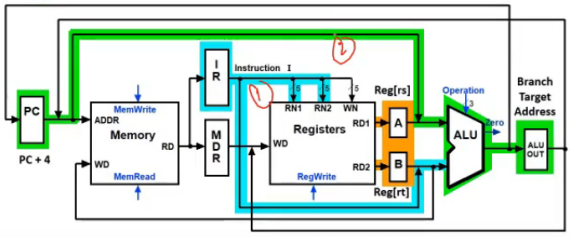
Multicycle Execution Step (1): Instruction Fetch

```
IR = Memory[PC];
PC = PC + 4;
```



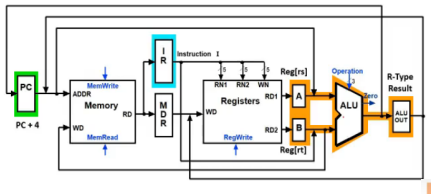
Multicycle Execution Step (2): Instruction Decode & Register Fetch

```
A = Reg[IR[25-21]];          (A = Reg[rs])
B = Reg[IR[20-15]];          (B = Reg[rt])
ALUOut = (PC + sign-extend(IR[15-0]) << 2)
```



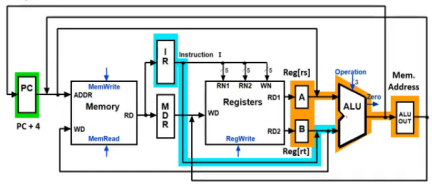
Multicycle Execution Step (3): ALU Instruction (R-Type)

```
ALUOut = A op B
```



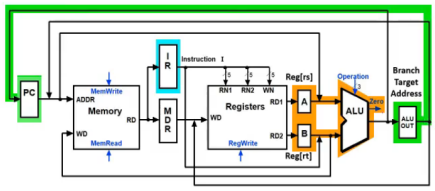
Multicycle Execution Step (3): Memory Reference Instructions

```
ALUOut = A + sign-extend(IR[15-0]);
```



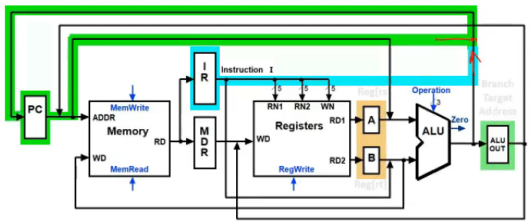
Multicycle Execution Step (3): Branch Instructions

```
if (A == B) PC = ALUOut;
```



Multicycle Execution Step (3): Jump Instruction

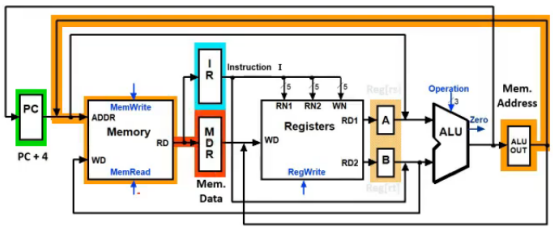
```
PC = PC[31-28] concat (IR[25-0] << 2)
```



IV

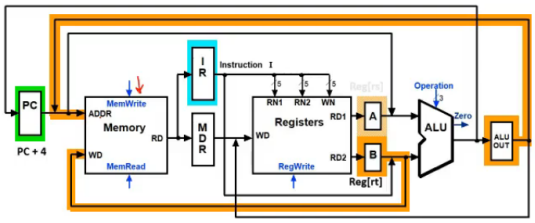
Multicycle Execution Step (4): Memory Access - Read (lw)

```
MDR = Memory[ALUOut];
```



Multicycle Execution Step (4): Memory Access - Write (sw)

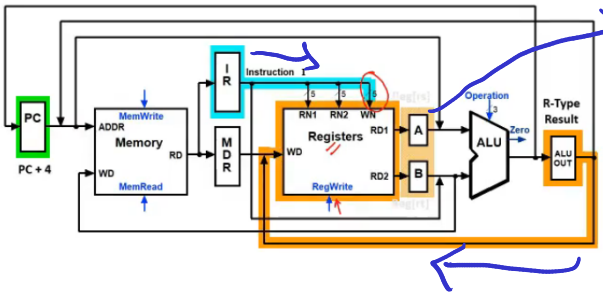
```
Memory[ALUOut] = B;
```



Multicycle Execution Step (4): ALU Instruction (R-Type)

```
Reg[IR[15:11]] = ALUOUT
```

then ALUOUT ← A op B

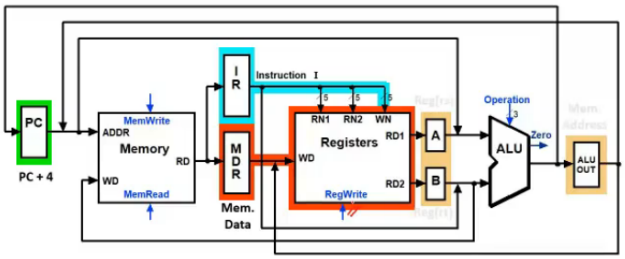


like

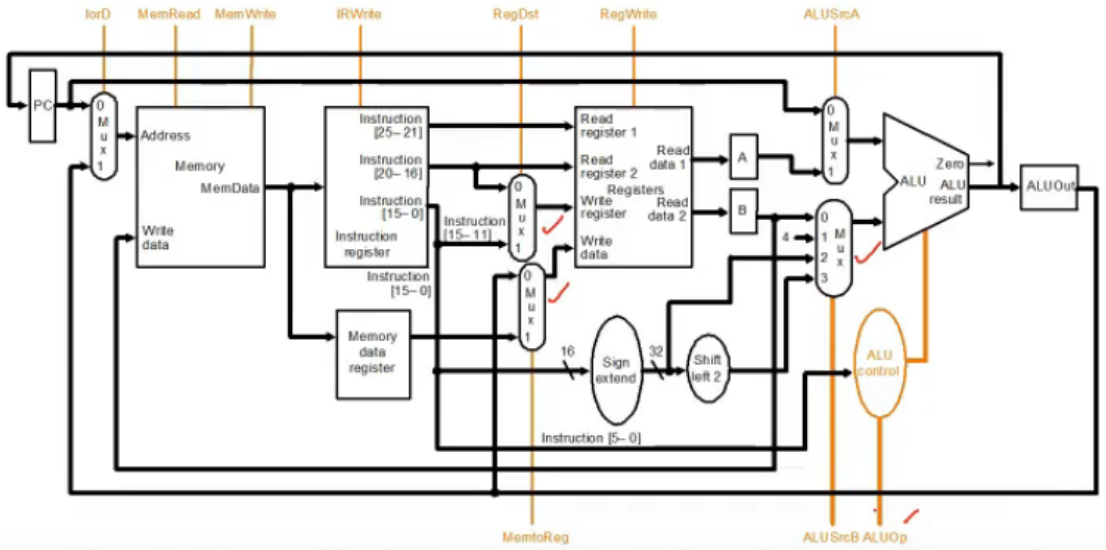
Multicycle Execution Step (5): Memory Read Completion (lw)

WB

```
Reg[IR[20-16]] = MDR;
```

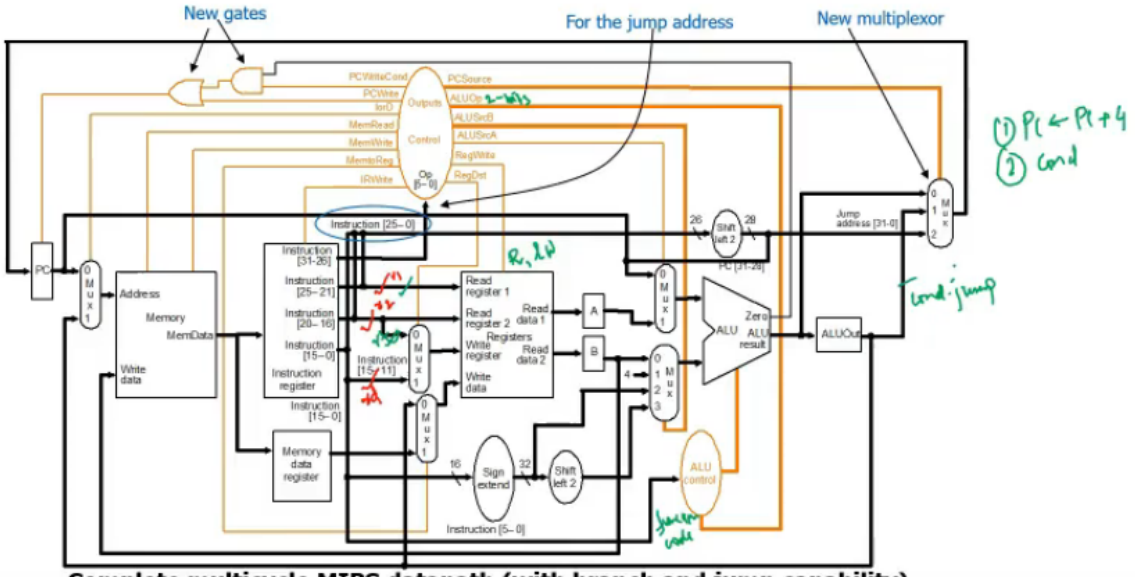


Multicycle Datapath with Control I



... with control lines and the ALU control block added – *not all* control lines are shown

Multicycle Datapath with Control II

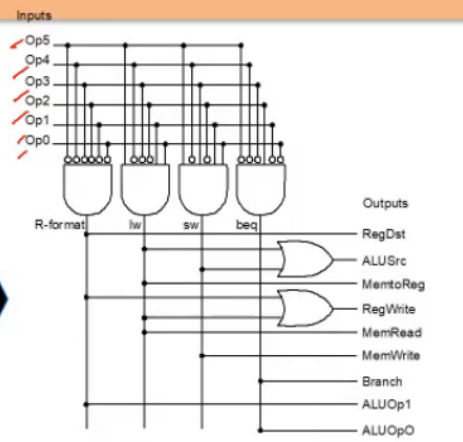


Complete multicycle MIPS datapath (with branch and jump capability) and showing the main control block and all control lines

Implementation: Main Control Block

	Signal name	R-format	lw	sw	beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	x	x
	ALUSrc	0	1	1	0
	MemtoReg	0	1	x	x
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOP2	0	0	0	1

Truth table for main control signals



Main control PLA (programmable logic array): principle underlying PLAs is that any logical expression can be written as a sum-of-products