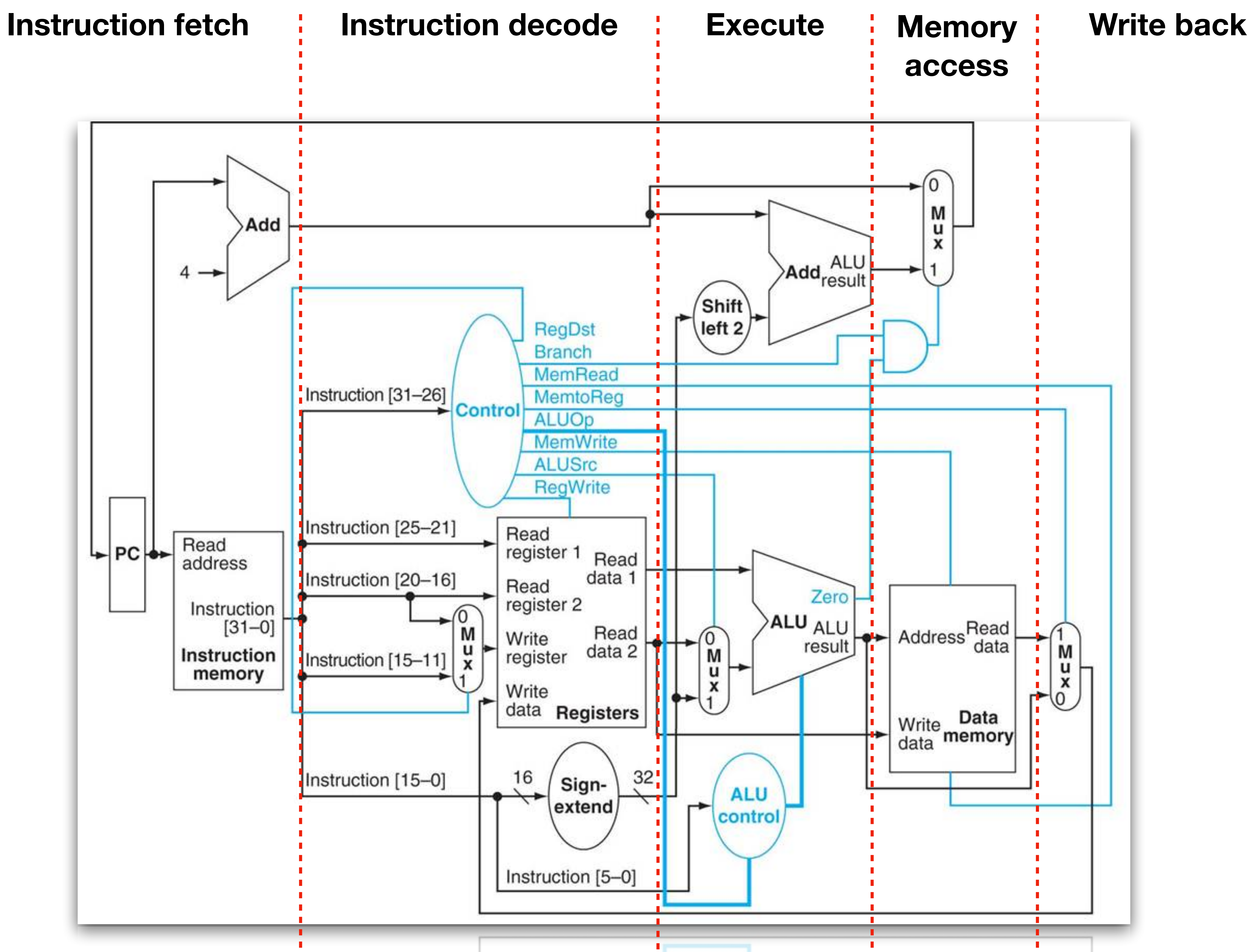
Datapath and Control

As a logic

Nitin


Instruction fetch
Instruction decode
Execute
Memory access
Write back
Add
4
PC
Read address
Instruction [31–0]
Instruction memory
Instruction [31–26]
Control
RegDst
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
Instruction [15–0]
Instruction [5–0]
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
16
Sign-extend
32
Shift left 2
Add
ALU result
Mux
0
1
ALU
Zero
ALU result
ALU control
Address
Read data
Write data
Data memory

Datapath

Relative codes

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

Memory Reference: `ALUout = A + SignExtend(IR[15:0])`

R-format Instruction: `ALUout = A op B`

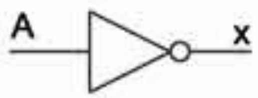
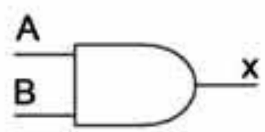
Branch: `if (A == B) then PC = ALUout`

Jump: `PC = PC[31:28] || (IR[25:0] << 2)`

```
A = RegFile[IR[25:21]]
B = RegFile[IR[20:16]]
ALUout = PC + SignExtend(IR[15:0]) << 2
```

```
MDR = Memory[ALUout]       # Load
Memory[ALUout] = B         # Store
Reg[IR[15:11]] = ALUout    # Write ALU result to register file
```

Logic Gates

Name	NOT	AND	NAND	OR	NOR	XOR	XNOR
Alg. Expr.	$\overline{A}$	AB	$\overline{AB}$	$A+B$	$\overline{A+B}$	$A \oplus B$	$\overline{A \oplus B}$
Symbol							
Truth Table	A | X 0 | **1** 1 | **0**	B A | X 0 0 | **0** 0 1 | **0** 1 0 | **0** 1 1 | **1**	B A | X 0 0 | **1** 0 1 | **1** 1 0 | **1** 1 1 | **0**	B A | X 0 0 | **0** 0 1 | **1** 1 0 | **1** 1 1 | **1**	B A | X 0 0 | **1** 0 1 | **0** 1 0 | **0** 1 1 | **0**	B A | X 0 0 | **0** 0 1 | **1** 1 0 | **1** 1 1 | **0**	B A | X 0 0 | **1** 0 1 | **0** 1 0 | **0** 1 1 | **1**

Logic gates are the basic building blocks of any digital system.

These logic gates are built using transistors, FETs.

Logics

Component	Logic	Input	Output
ALU adder for PC	XOR gate	PC address, +4	PC+4
Multiplexors	AND, OR - SOP	2-datalines, 1-control signal	0/1
PCSrc	AND	Branch, Zero	0/1
ALU Control	AND, OR, XOR	funct, ALUOp	4-bits

Some more logics

NOT, AND, NOR

- In pipelining, to allow forwarding, we use flip-flops as buffers.

- Instruction memory is a tautology (T). i.e., is operates always (fetches instructions), unlike Data Memory which operates on for I-type instructions.

- A bitwise-operator 'shift-left 2' is used. (MIPS instructions are 32 bits = 4 bytes, so the branch offset is specified as a multiple of 4, i.e. a branch offset of 1 = 4 bytes).

- ~MUX (0) = MUX(1). (As there are only two inputs for MUX, both are compliment to each other. (NOT gate)

- MUX(0) and MUX(1) is a contradiction.

Conditional statements

- Read register 2 $\longrightarrow$ Read data 2.

i.e., Data from register 2 is read only if it is activated from the MUX.

- Memory Data $\longleftrightarrow$ (MemRead $\vee$ MemWrite)
- PCSrc $\longleftrightarrow$ (Branch $\wedge$ 'Zero')
- R-type $\longleftrightarrow$ (RegDst == 1)

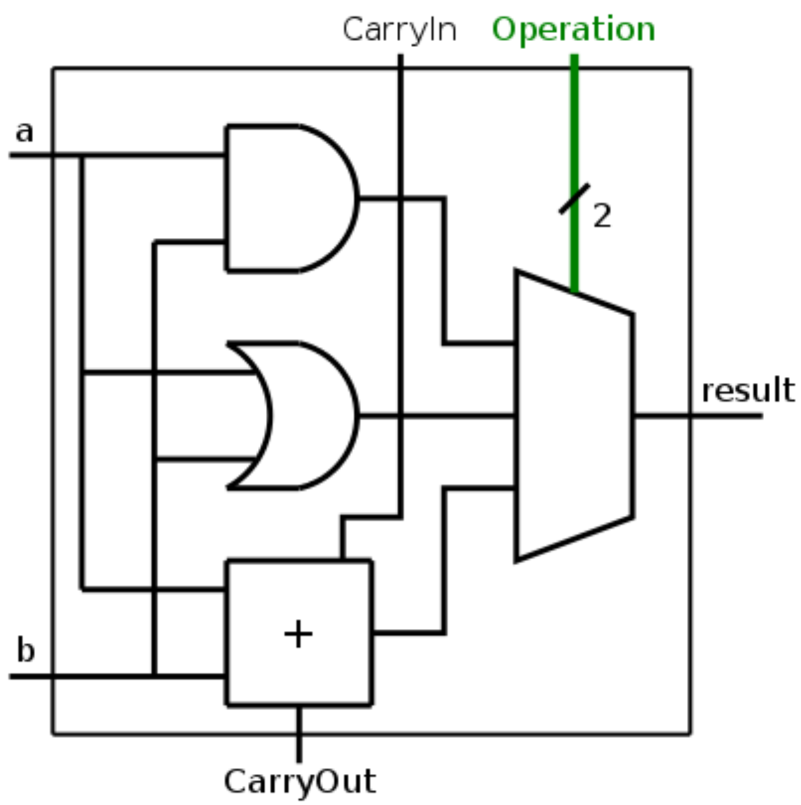
Predicate logics

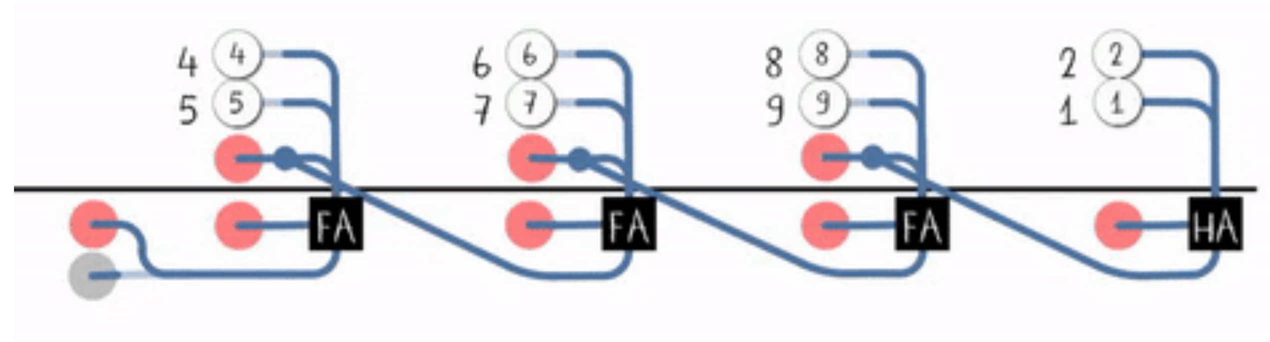
- A predicate is an expression of one or more variables determined on some specific domain which turns out to be a proposition for some value of variable in the domain.

Predicate	Variable	Domain	Range
a is R-type	a	add, sub, lw, sw, beq, slt, sll,. etc	True/False
Destination register is b	b	rt, rd	True/False
Next PC value is c	c	PC+4, jump address	True/False

- Therefore a predicate is a propositional function.

Adder





Ripple Carry Adder

Adder is a combination of XOR, AND, OR gates.

In 32-bit adder, $C_{out} = A.B + C_{in}.(A + B)$ for i^{th} adder = C_{in} for $(i+1)^{th}$ adder.

Quantifiers

- Every data item can be quantified.
- In this datapath, we define 3 quantifiers.
  1. Universal
  2. Existential
  3. Unique

Expression	Quantifier used	Statement	Domain
$\forall_{ALUOp = 10}$ **R-type**	Universal	For all ALUOp as 01, the instruction is R-type.	00, 01, 10
$\exists_{ALUOp = 00}$ **I-type**	Existential	For some ALUOp as 00, the instruction is I-type.	00, 01, 10
$\exists!_{MUX}$ **rt**	Unique	There exists only one valid of MUX for which the destination register is rt.	0, 1

Functions

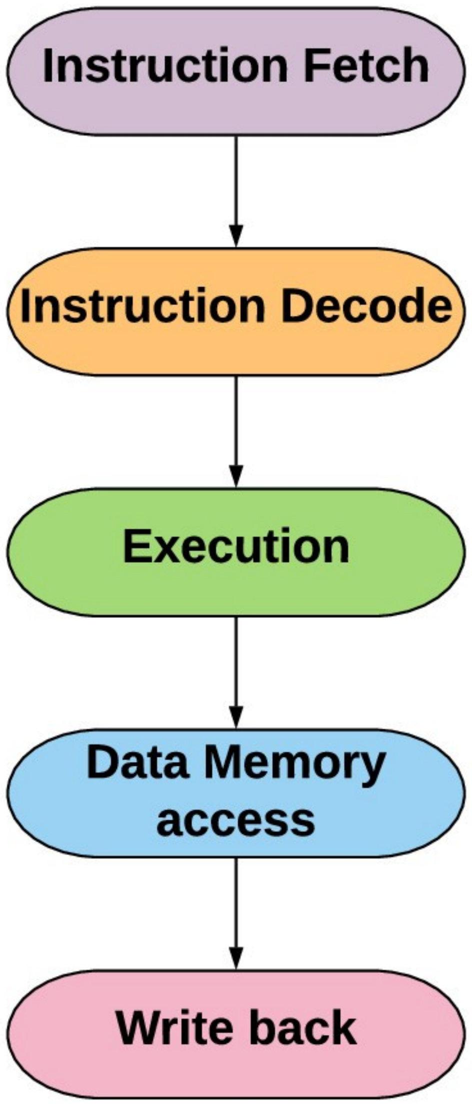
Component : Input → Output

Component	Input	Output
PC ALU adder	PC	PC + 4
Control Unit	opcode	Control lines
Register file	Instruction[11:25]	Read data 1, Read data 2
Instruction memory	PC address	Instruction[0:31]
Sign-extend	16-bits	32-bits
ALU	Read data1, Read data 2/sign-extend	Arithmetic output
ALU Control	ALUOp, funct	4-bits
Data memory	Address/value	Data/(x)

Graphs

- Datapath for each type of instruction can be represented as directed graph.
- If the complete datapath is represented as graph, then each instruction can be represented as subgraph.
- The subgraphs can form either trees, circuits or cycles.

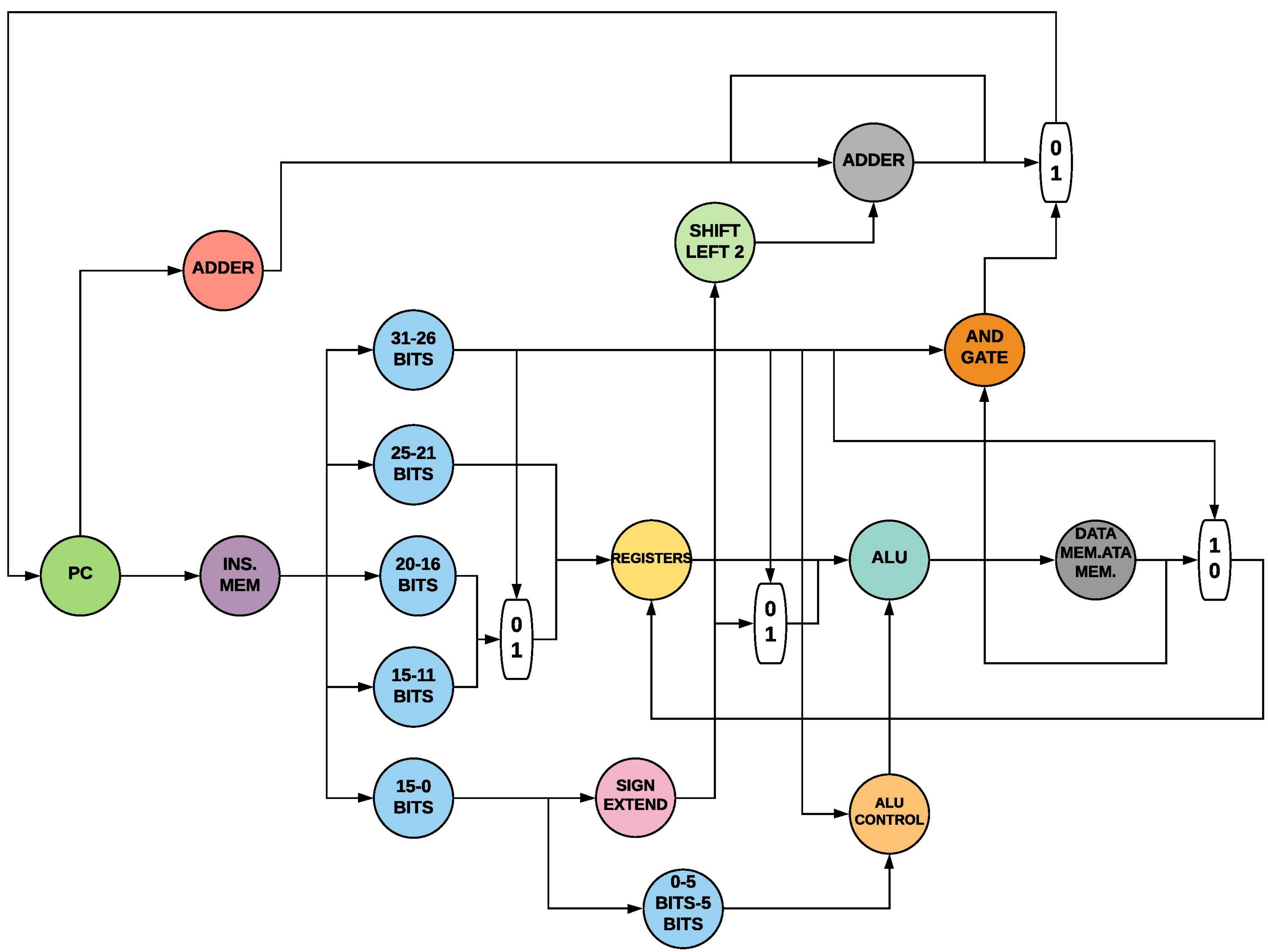
Steps in Datapath



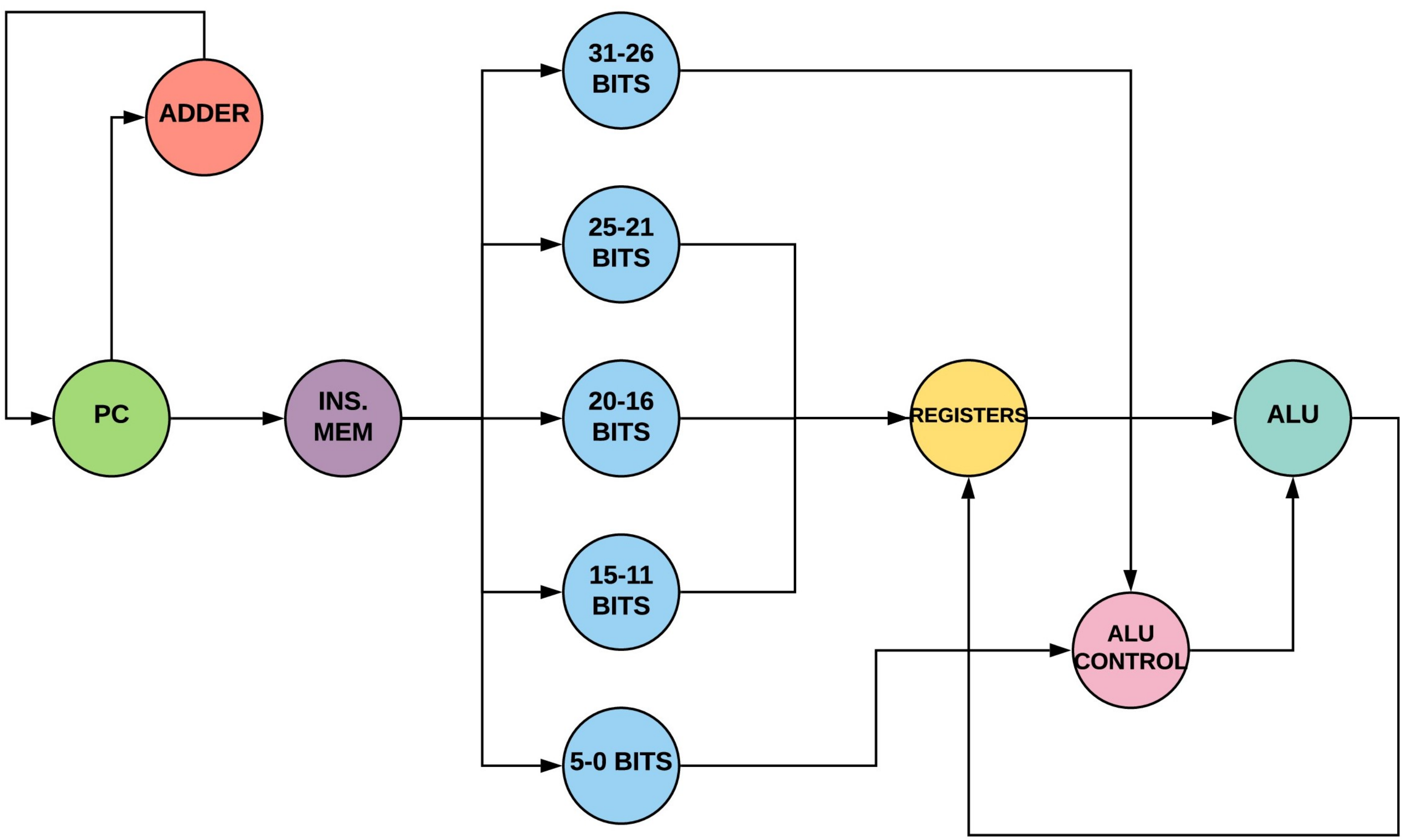
1. Fetch instruction from memory.
2. Read register while decoding the instruction.
3. Execute the operation or calculate the address.
4. Access an operand in data memory.
5. Write the result into register.

NOTE: This is a tree. A step is done only after the completion of previous step.

Datapath as Graph



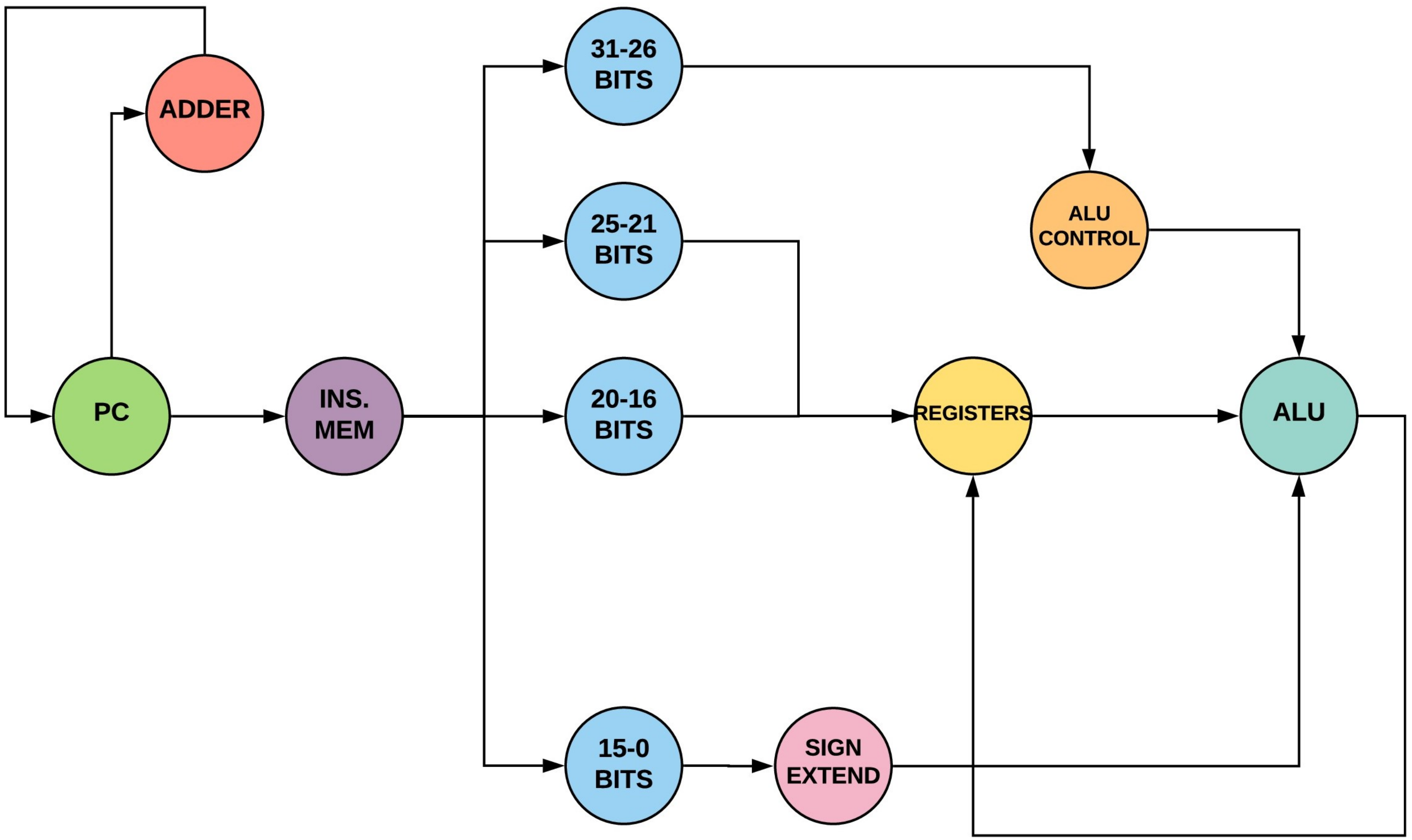
ADD Instruction



This graph is obtained by removing following vertices:

- Adder
- Shift left 2
- And gate
- Data memory
- Sign extend

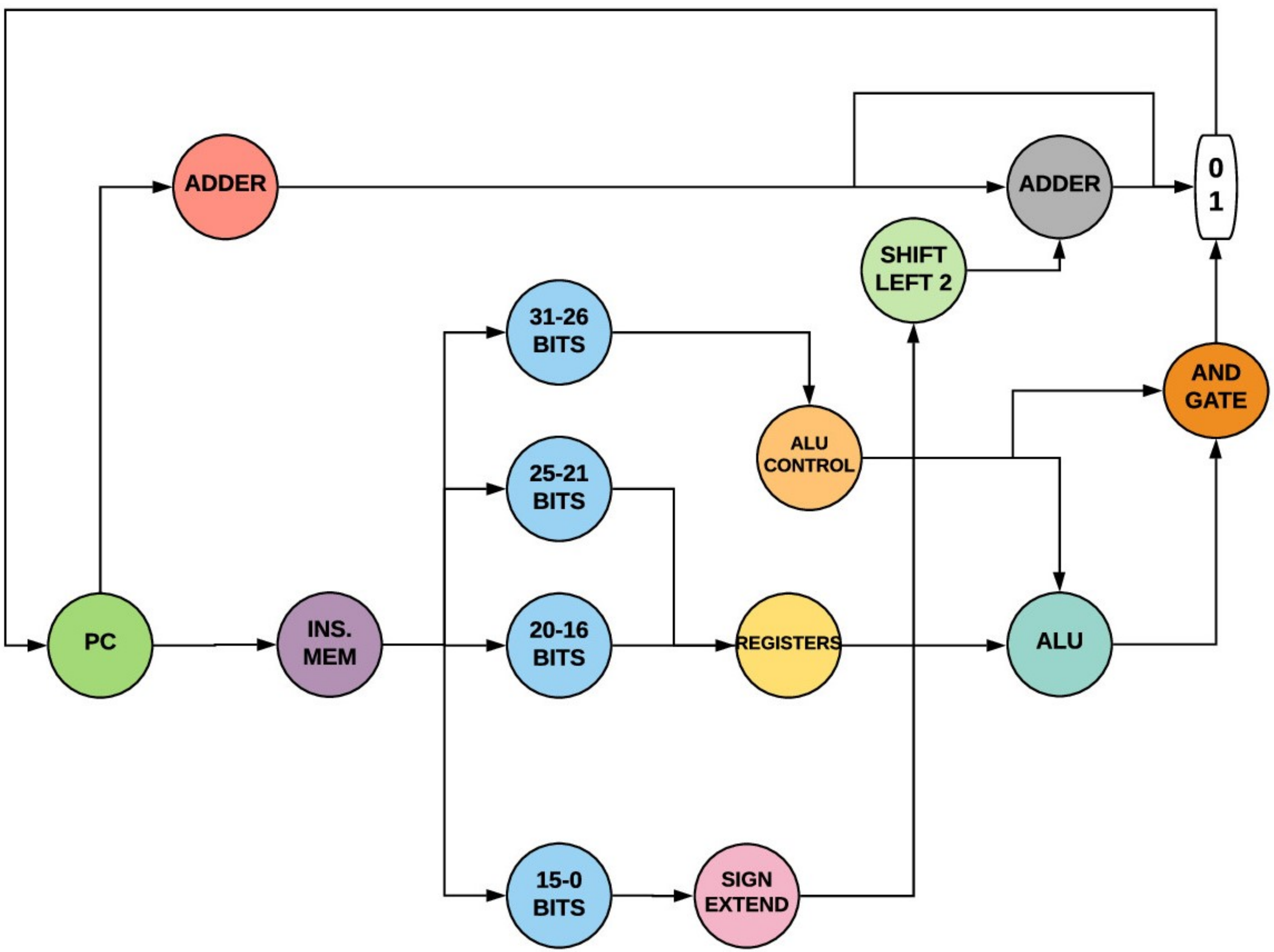
ADDI Instruction



This graph is obtained by removing following vertices:

- Adder
- Shift left 2
- And gate
- Data memory

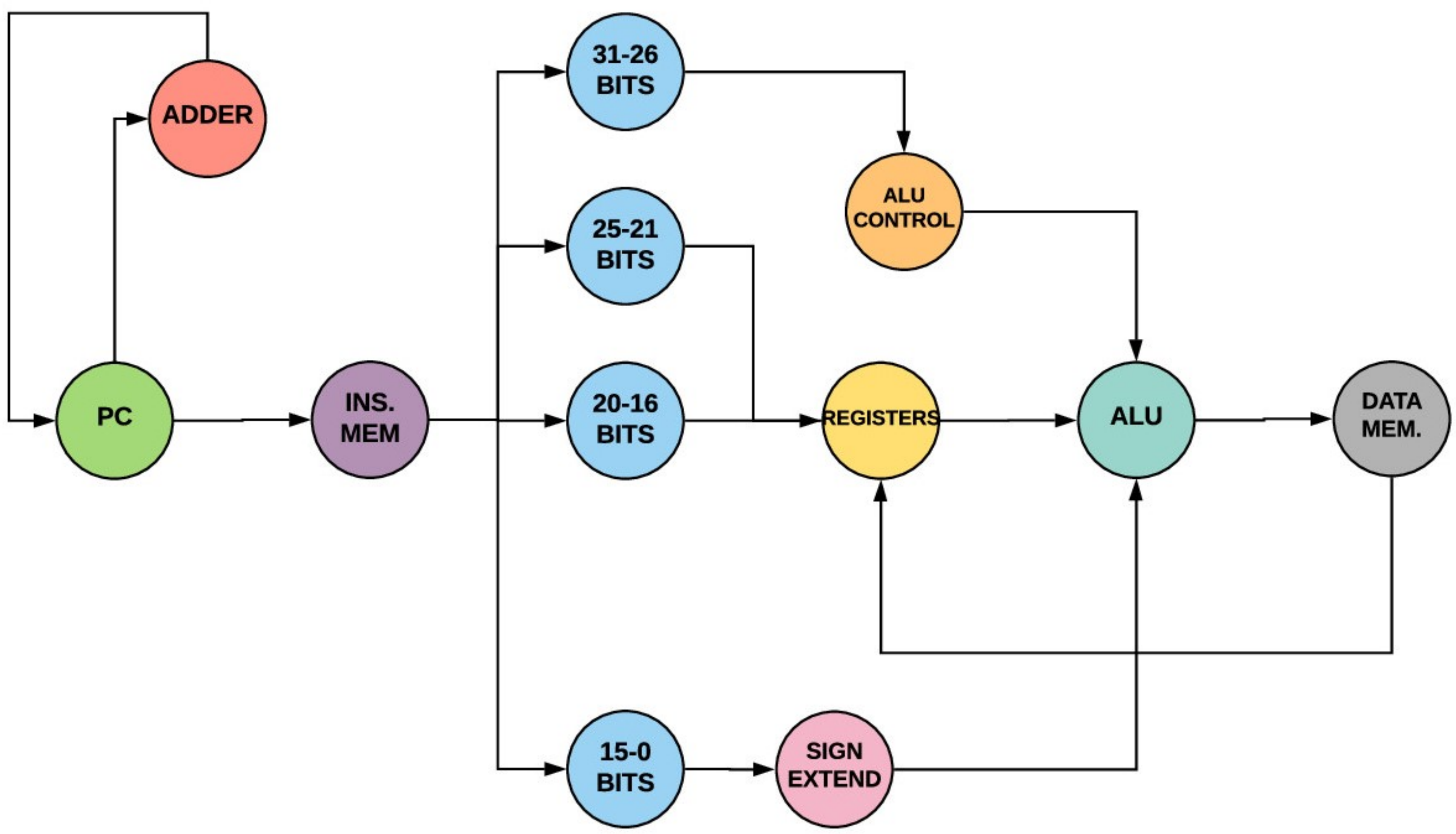
BEQ Instruction



This graph is obtained by removing:

- Data memory vertex

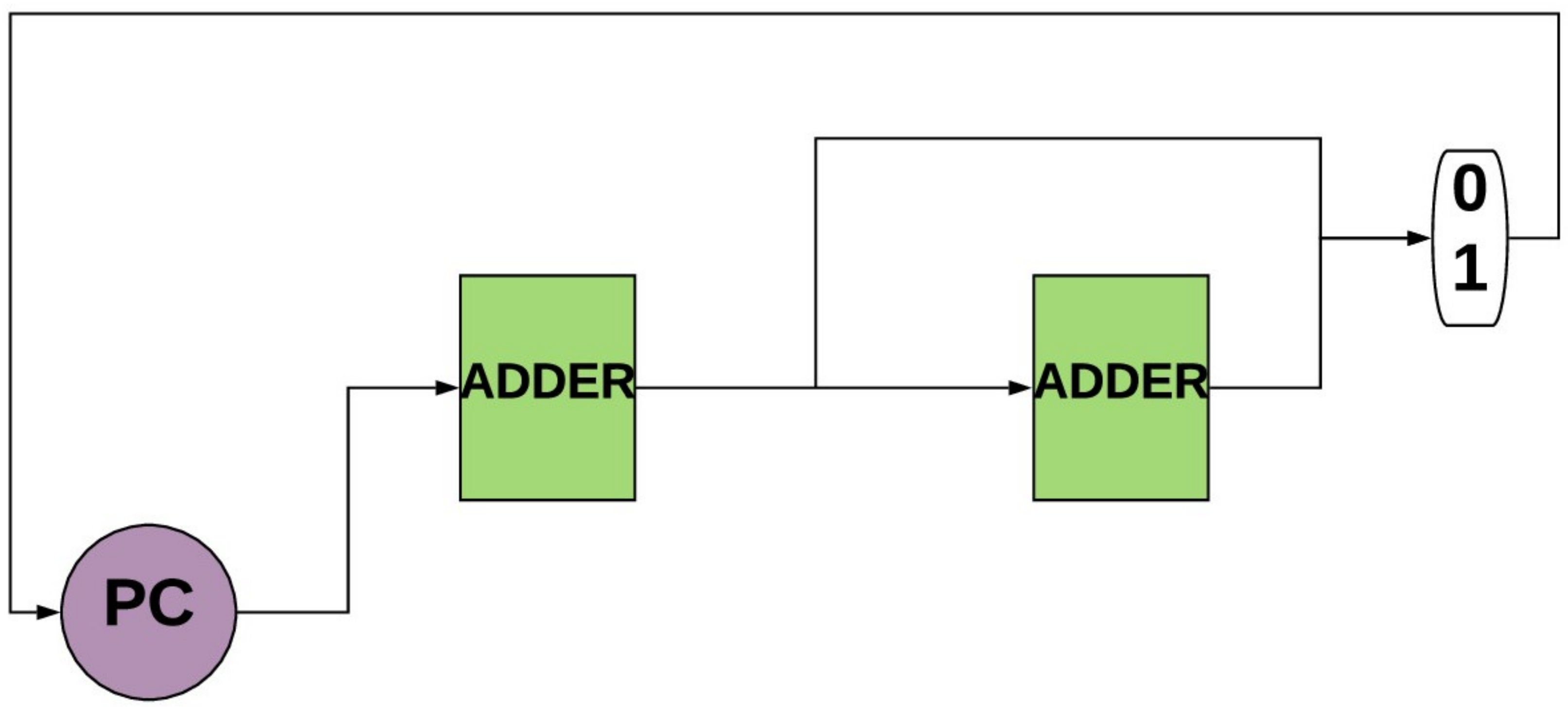
LW Instruction



This graph is obtained by removing following vertices:

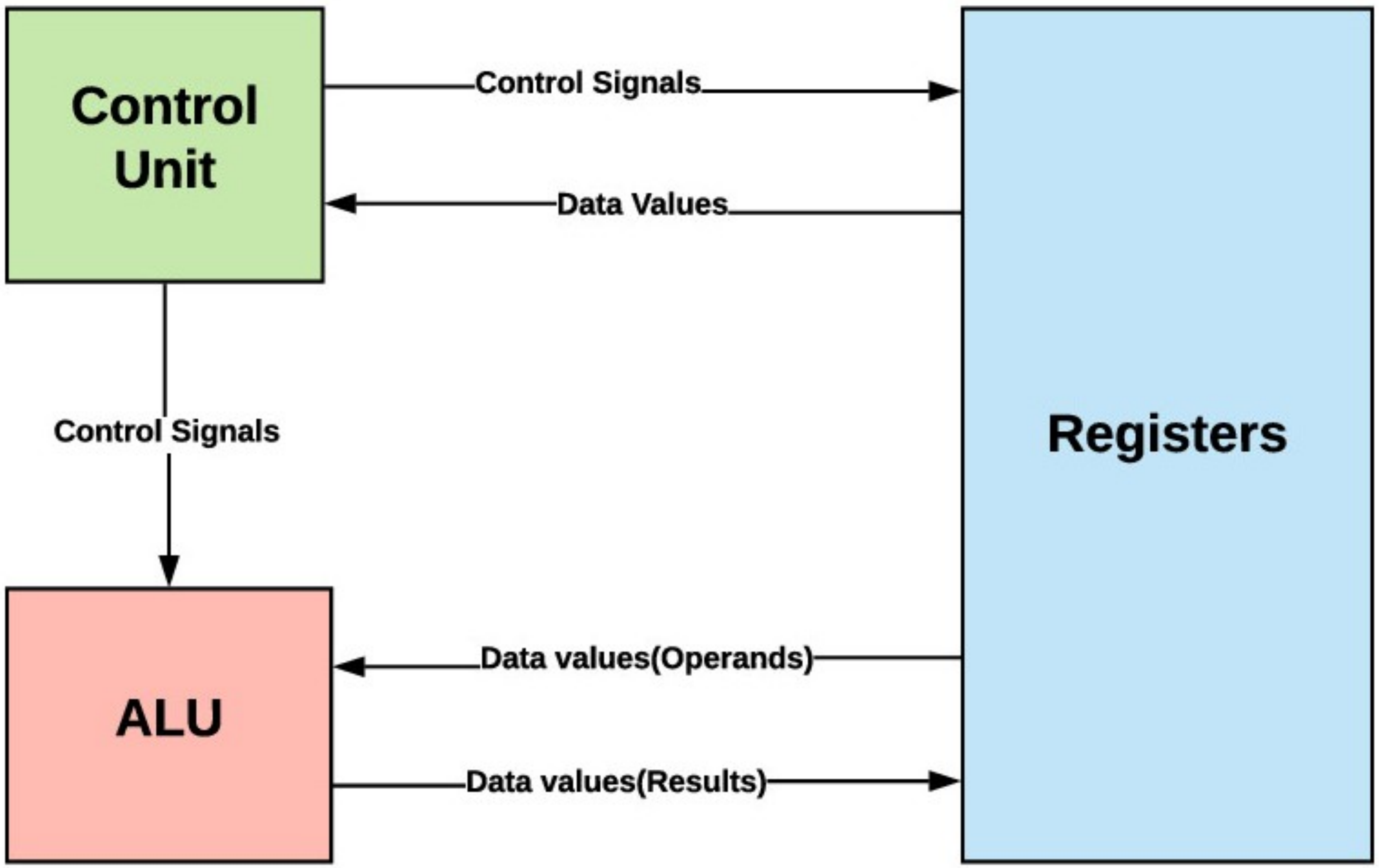
- Adder
- Shift left 2
- And gate
- [0-5] bits

PC



- PC graph forms a circuit
- Path : PC — adder — adder — MUX — PC
- It doesn't have any repeated edges.
- It is also a cycle since it doesn't have repeated vertices except the initial and the final vertices (PC).

Multigraph


Control Unit
Control Signals
Data Values
Registers
Control Signals
ALU
Data values(Operands)
Data values(Results)

Control signals as tree