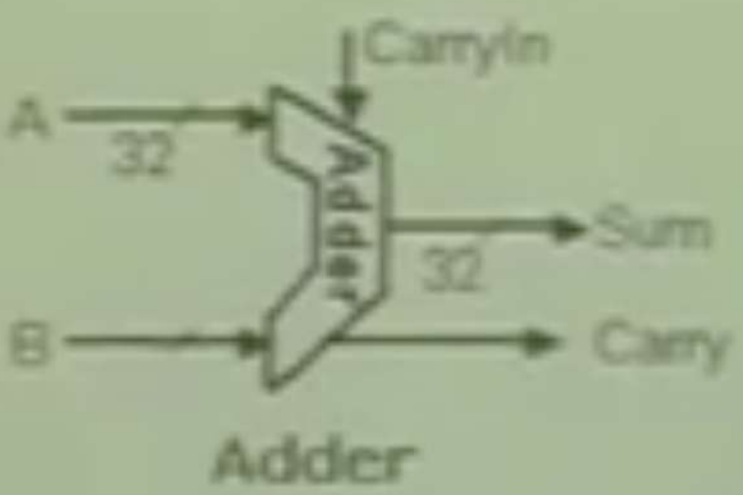
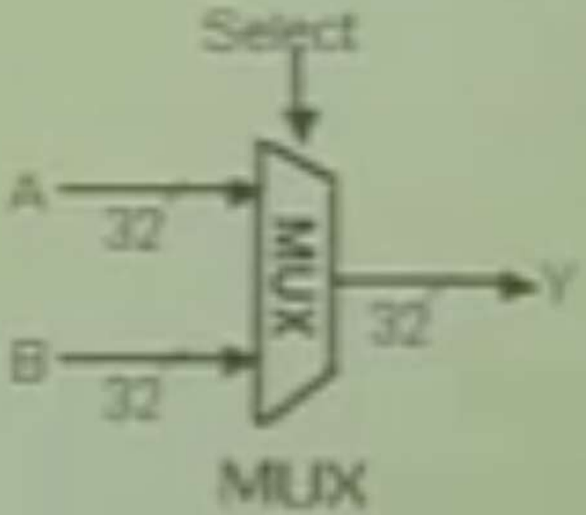
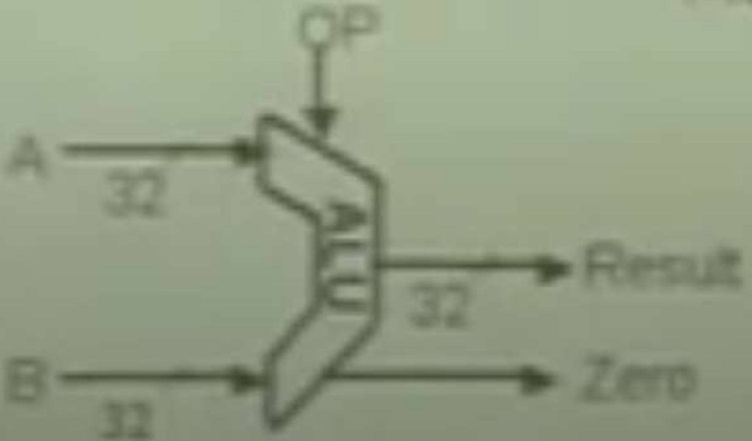
Combinational Elements



Adder



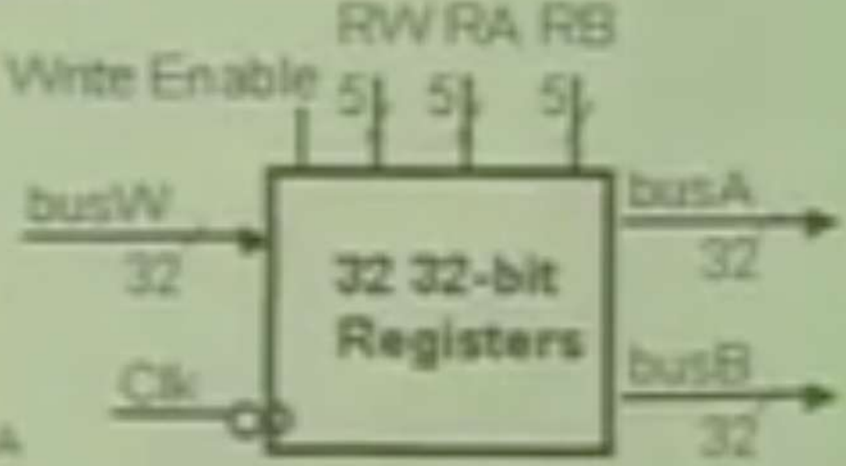
MUX



ALU

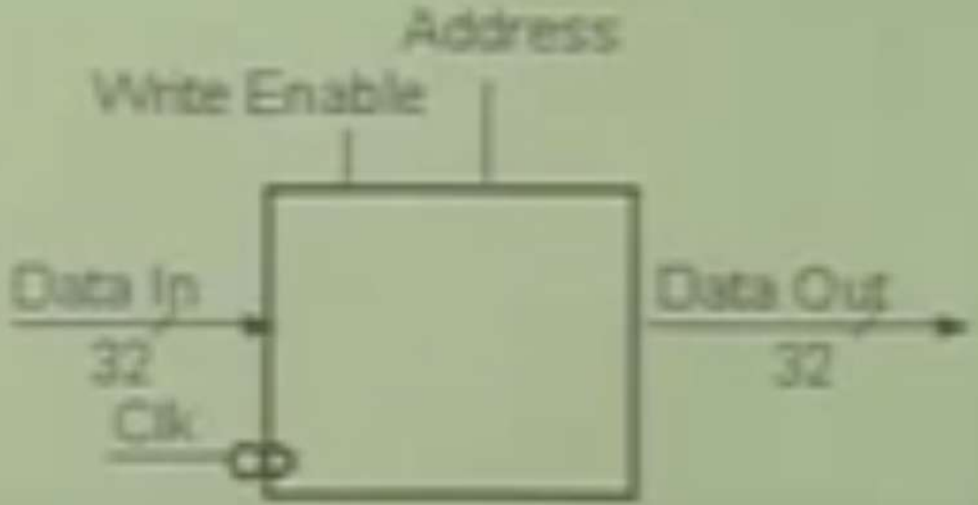
Storage Element: Reg File

- Register File consists of 32 registers
  - Two 32 bit output busses
    - busA and busB
  - One 32 bit input bus
    - busW
  - Register 0 hard wired to value 0
  - Register selected by
    - RA selects register to put on busA
    - RB selects register to put on busB
    - RW selects register to be written via busW when Write Enable is 1
  - Clock input (CLK)
    - CLK input is a factor only for write operation
    - During read, behaves as combinational logic block
      - RA or RB stable => busA or busB valid after "access time"
      - Minor simplification of reality



Storage Element: Memory

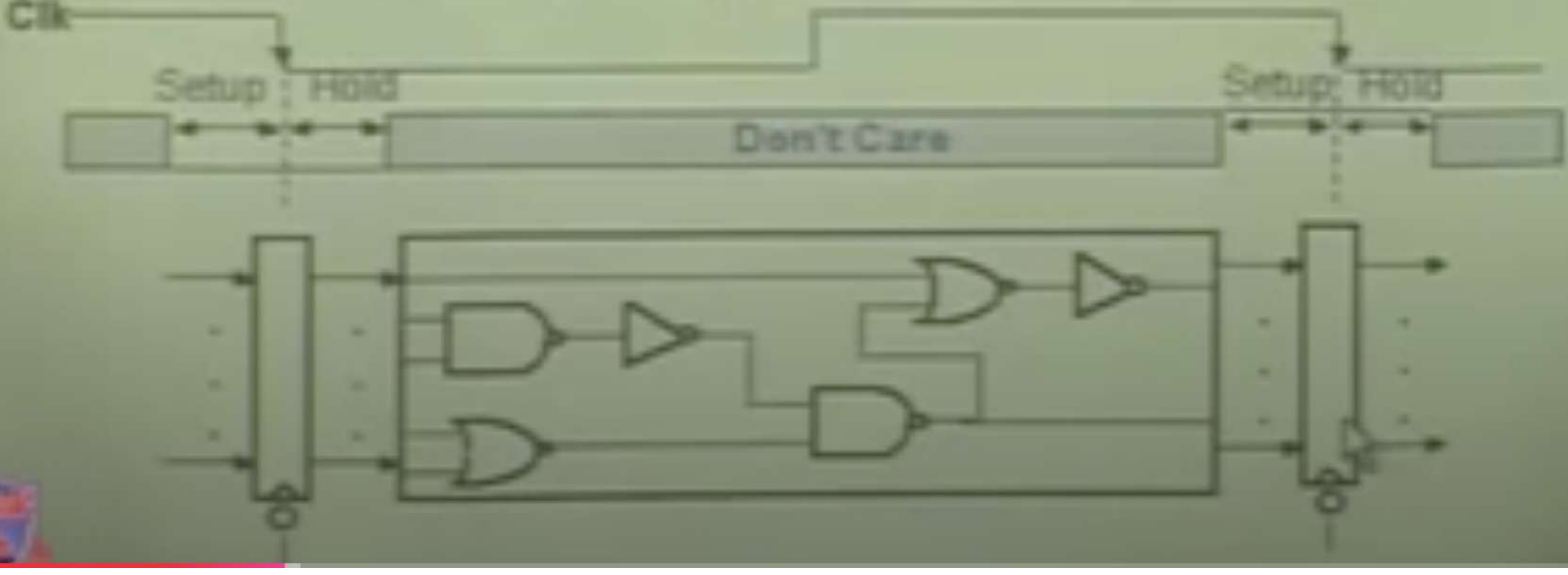
- Memory
  - One input bus: Data In
  - One output bus: Data Out
  - Address selection
    - Address selects the word to put on Data Out
    - To write to address, set Write Enable to 1
  - Clock input (CLK)
    - CLK input is a factor only for write operation
    - During read, behaves as combinational logic block
      - Valid Address → Data Out valid after "access time"
      - Minor simplification of reality



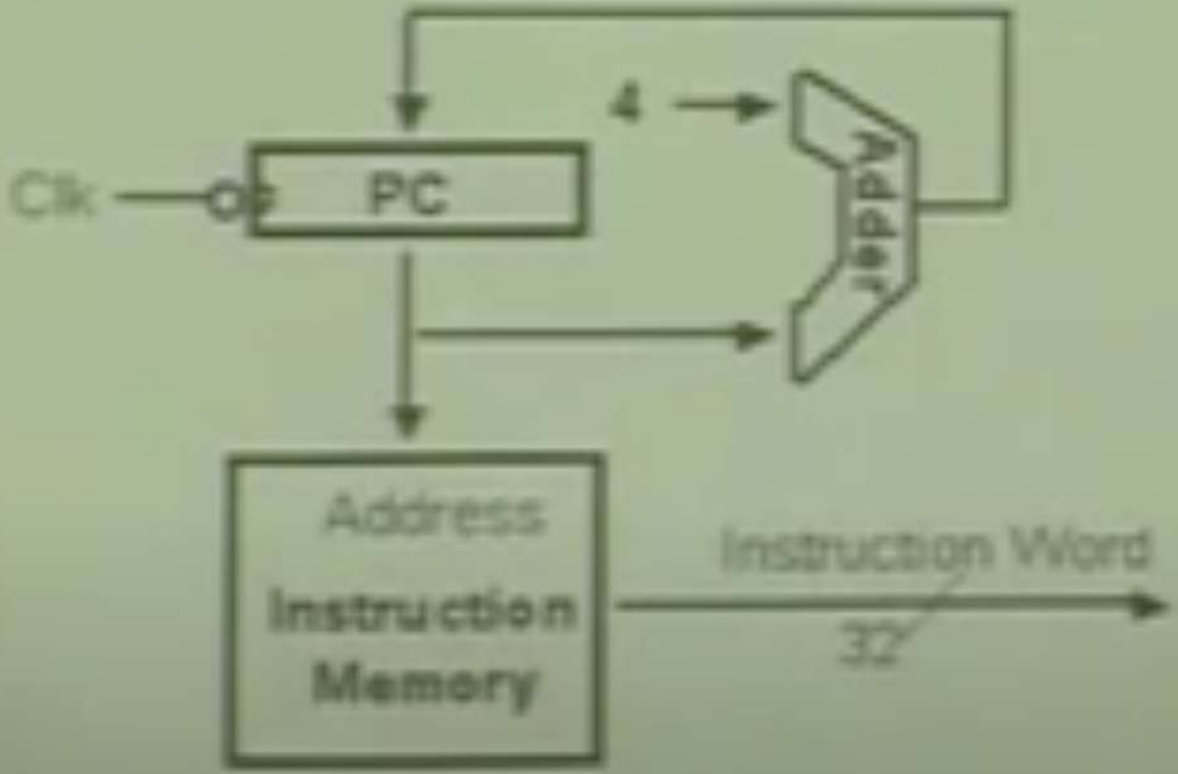
Some Logic Design…

- **All storage elements have same clock**
  - Edge-triggered clocking
  - "Instantaneous" state change (simplification!)
  - Timing always work if the clock is slow enough

Cycle Time = Clk-to-Q + Longest Delay + Setup + Clock Skew



Datapath: IF Unit



Add RTL

- Add instruction

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

```
Mem[PC];                        Fetch instruction from memory
R[rd] <- R[rs] + R[rt];         Add operation
PC <- PC + 4;                   Calculate next address
```

Bits	6	5	5	5	5	6
	OP=0	rs	rt	rd	sa	funct
		first source register	second source register	result register	shift amount	function code (=[illegible])

Sub RTL

- Sub instruction

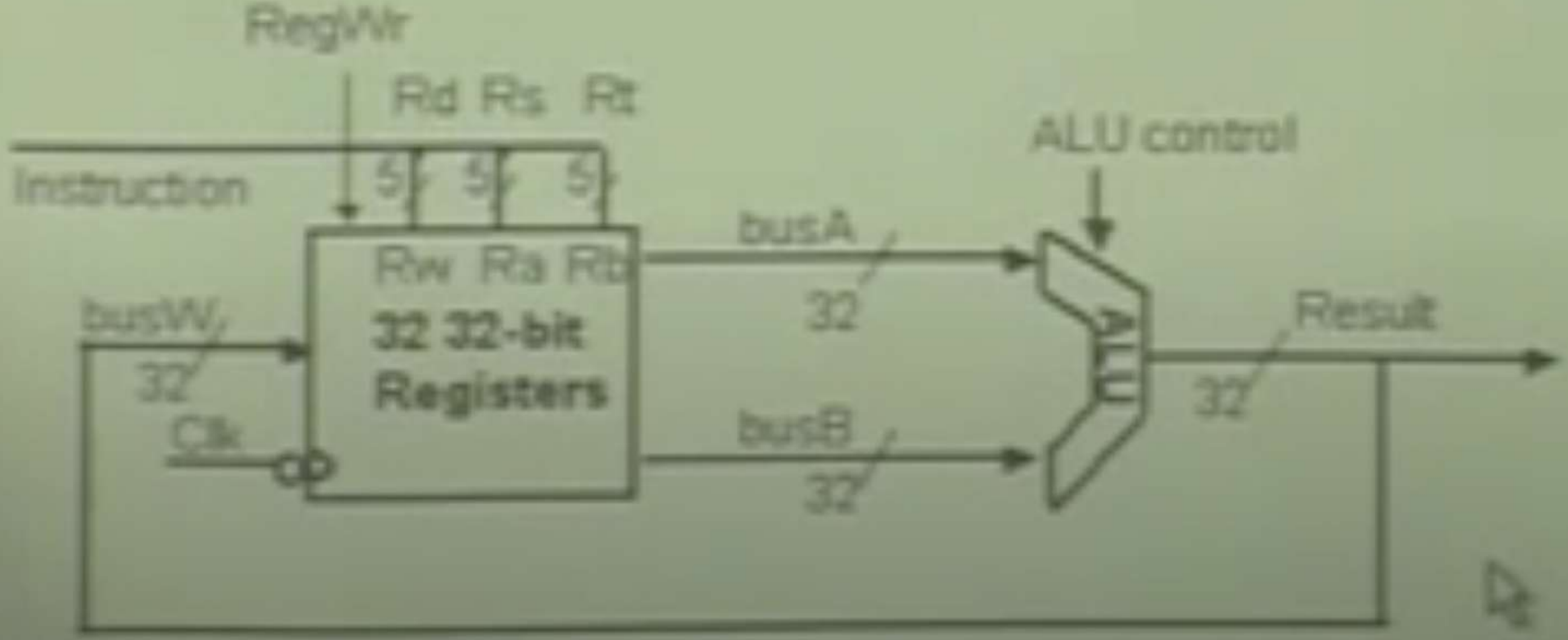
```
sub rd, rs, rt
```

```
Mem[PC];                          Fetch instruction from memory
R[rd] <- R[rs] - R[rt];           Sub operation
PC <- PC + 4;                     Calculate next address
```

Bits	6	5	5	5	5	6
	OP=0	rs	rt	rd	sa	funct
		first source register	second source register	result register	shift amount	function code (=[illegible])

Datapath: Reg/Reg Ops

- `R[rd] <- R[rs] op R[rt];`
  - ALU control and RegWr based on decoded instruction
  - Ra, Rb, and Rd from rs, rt, rd fields



OR Immediate RTL

- OR Immediate instruction

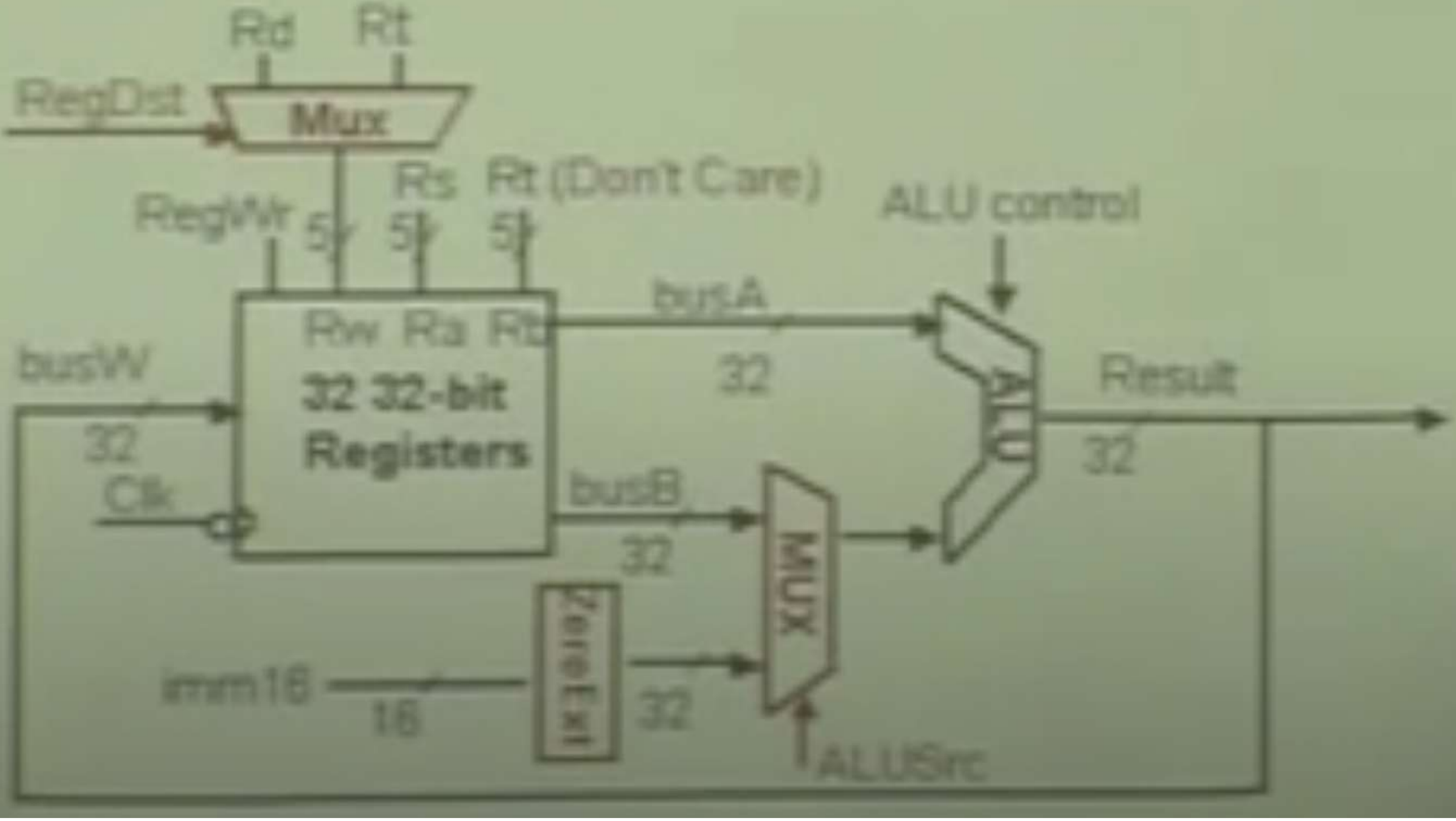
```
ori rt, rs, imm
```

```
Mem[PC];                                  Fetch instruction from memory
R[rt] <- R[rs] OR ZeroExt(imm);
                                          OR operation with Zero-Extend
PC <- PC + 4;                             Calculate next address
```

Bits	6	5	5	16
	OP	rs	rt	imm
		first source register	second register (dest)	immediate

Datapath: Immediate Ops

- Rw set by MUX and ALU B set as busB or ZeroExt(imm)
- ALUsrc and RegDst set based on instruction



Load RTL

- Load instruction

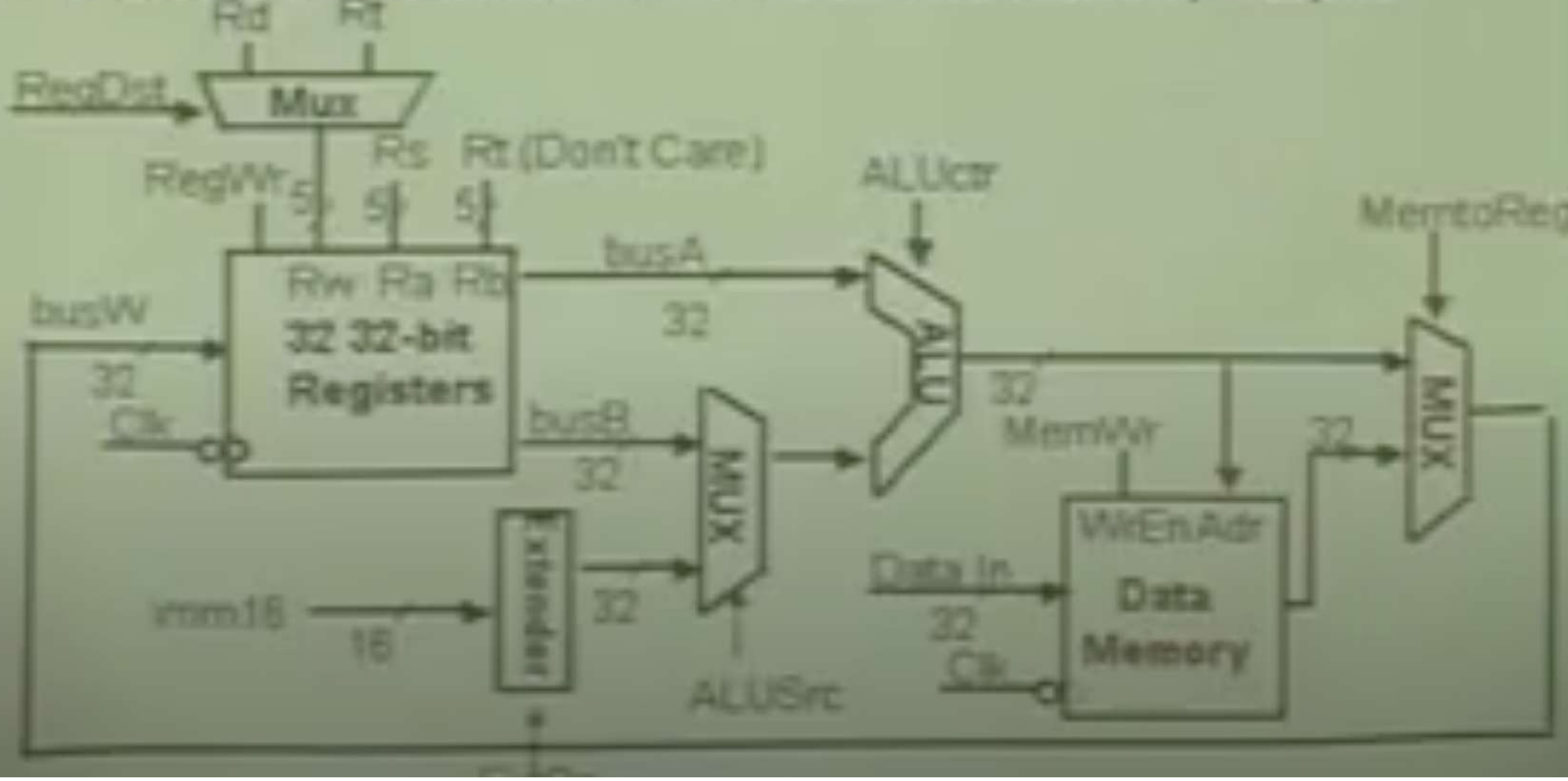
```
lw rt, rs, imm
```

```
Mem[PC];                        Fetch instruction from memory
Addr <- R[rs]+SignExt(imm);     Compute memory addr
R[rt] <- Mem[Addr];             Load data into register
PC <- PC + 4;                   Calculate next address
```

Bits	6	5	5	16
	OP	rs	rt	imm
		first source register	second register (dest)	immediate

Datapath: Load

- Extender handles sign vs. zero extension of immediate
- MUX selects between ALU result and Memory output



Store RTL

- Store instruction

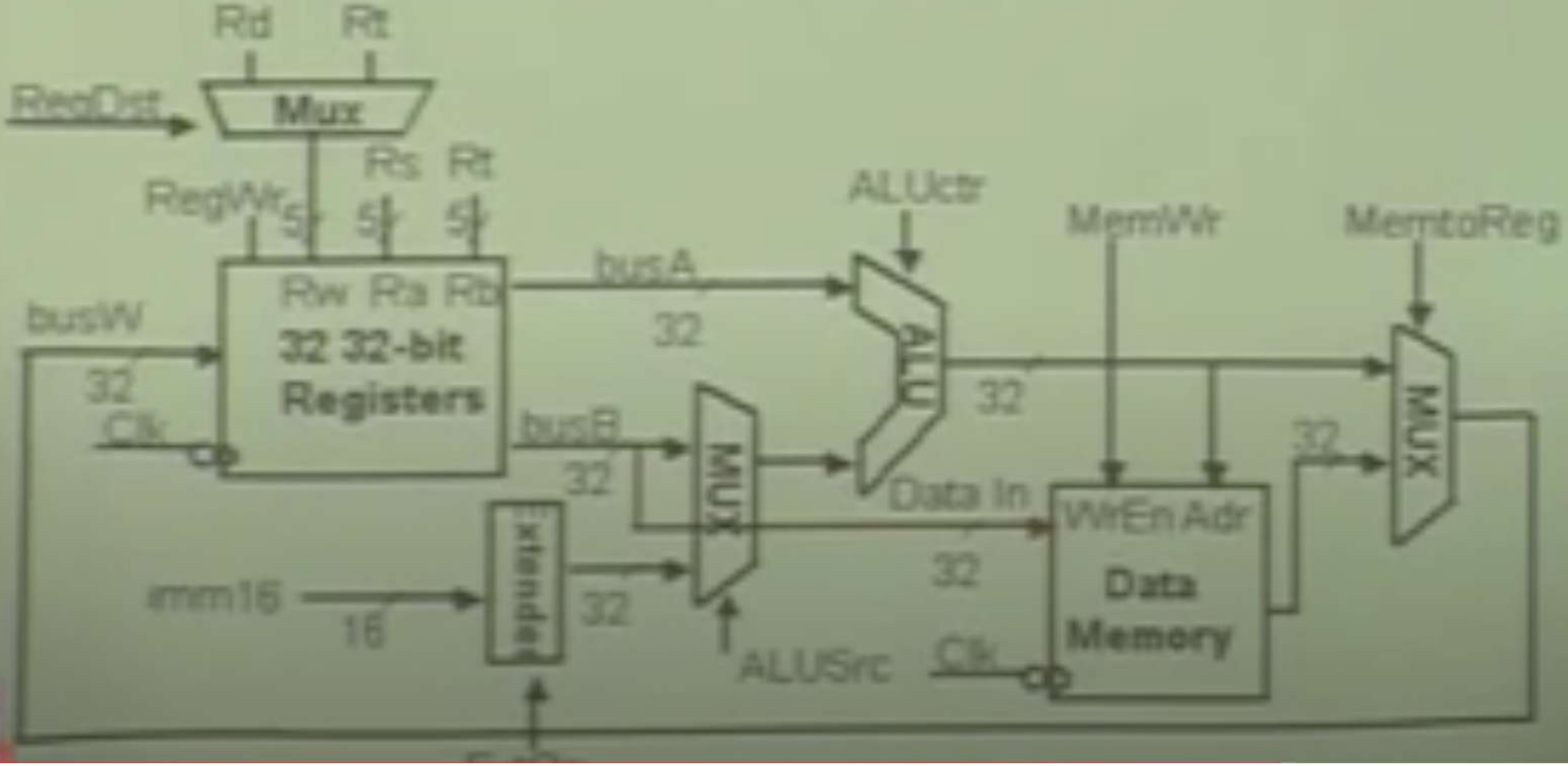
```
sw rt, rs, imm

    Mem[PC];                          Fetch instruction from memory
    Addr <- R[rs]+ SignExt(imm);  Compute memory addr
    Mem[Addr] <- R[rt];               Load data into register
    PC <- PC + 4;                     Calculate next address
```

Bits	6	5	5	16
	OP	rs	rt	imm
		first source register	second source register	immediate

Datapath: Store

- Register rt is passed on busB into memory
- Memory address calculated just as in `lw` case



Branch RTL

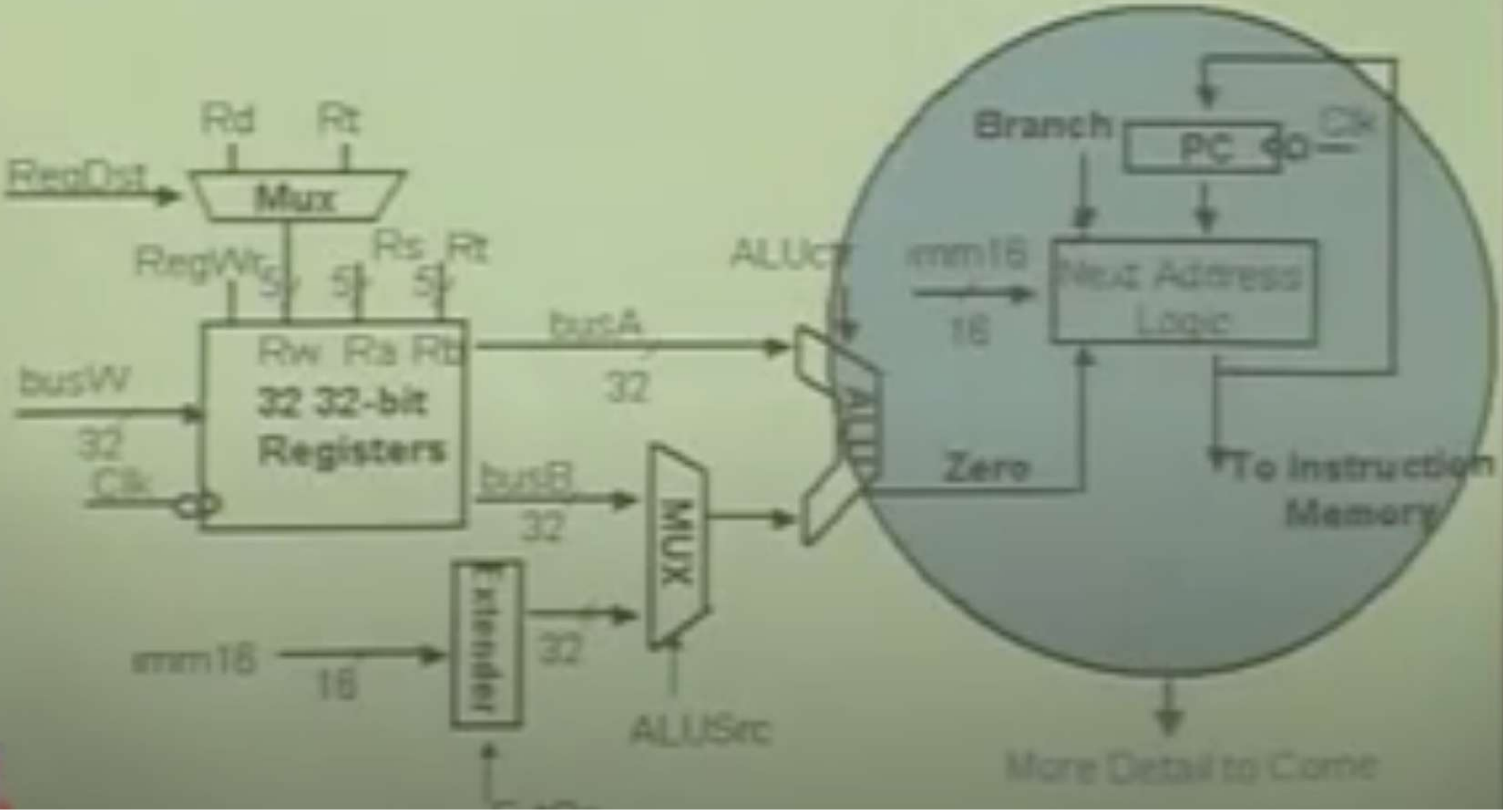
- Branch instruction

```
beq rs, rt, imm

Mem[PC];                                 Fetch instruction from memory
Cond <- R[rs] - R[rt];                   Calculate branch condition
if (Cond eq 0)                           Test if equal
     PC <- PC + 4 +
            SignExt(imm)*4;              Calculate PC Relative address
else
     PC <- PC + 4;                       Calculate next address
```

Bits	6	5	5	16
	OP	rs	rt	imm
		first source	second source	immediate

Datapath: Branch



Rd
Rt
RegDst
Mux
RegWr
Rs
Rt
5
5
5
Rw
Ra
Rb
32 32-bit Registers
busW
32
Clk
busA
32
busB
32
MUX
Extender
imm16
16
32
ALUSrc
ALUctr
ALU
Zero
Branch
PC
Clk
imm16
16
Next Address Logic
To Instruction Memory
More Detail to Come