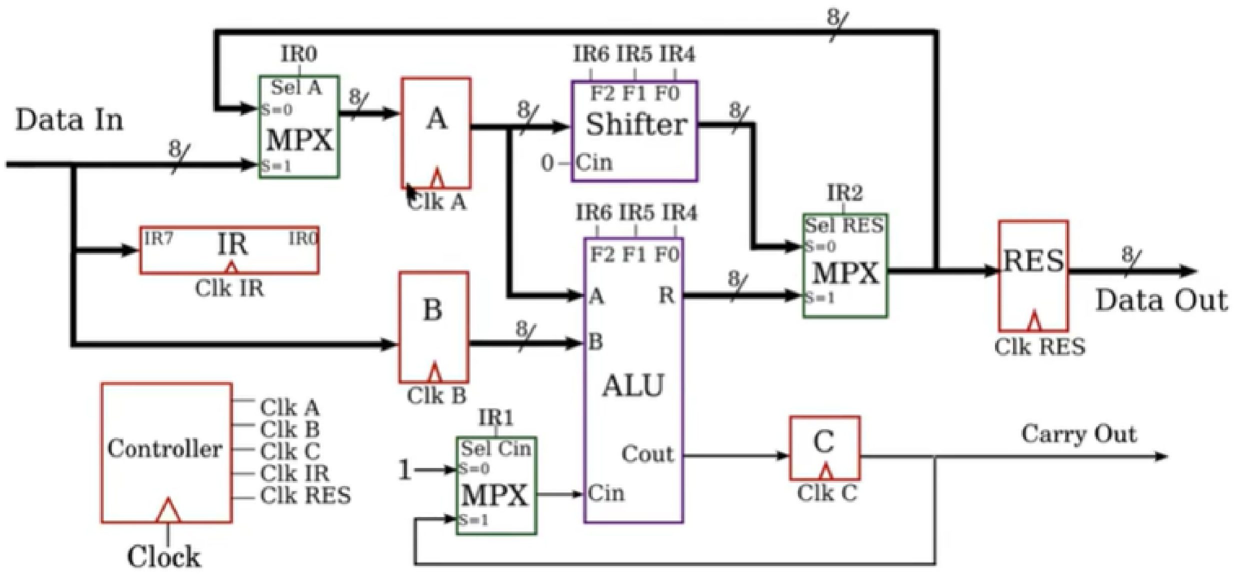
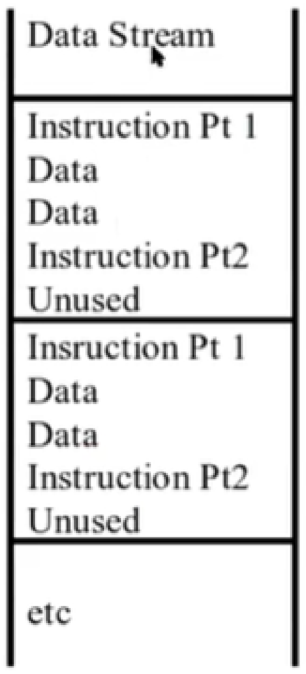
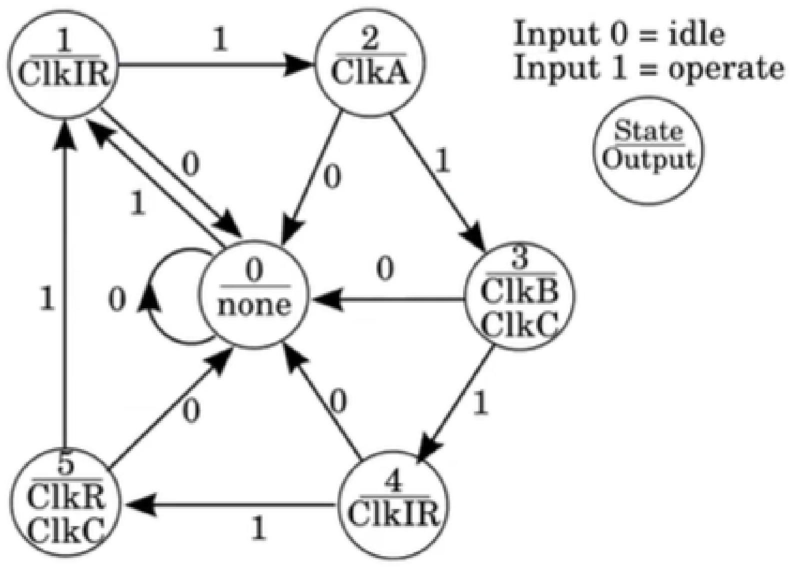
# L18 - Designing a Central Processor Unit 1 The Architecture

The Manual Processor



Execution Cycle

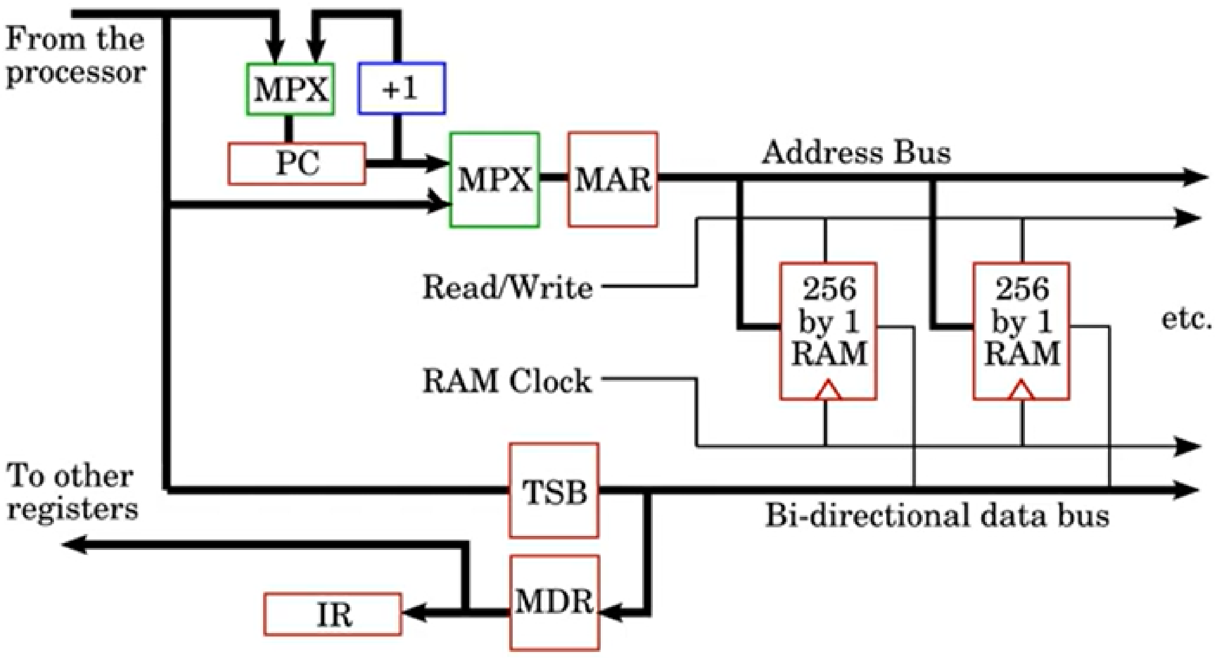




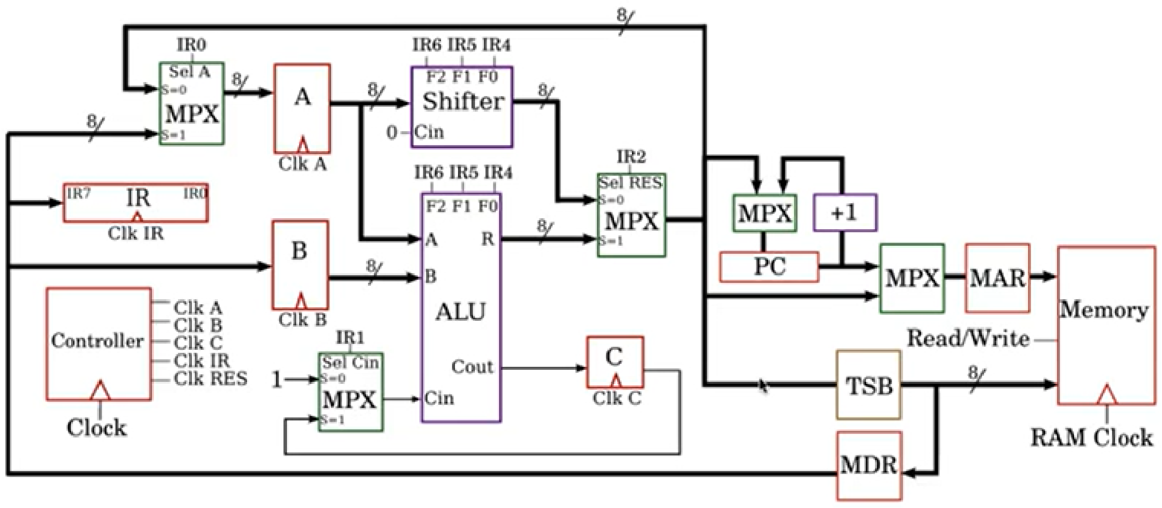
## Executing Instructions

1. Set the multiplexers so that the correct registers are connected together
2. Set the arithmetic hardware to compute the correct function
3. Provide a clock falling edge to the registers that are to change

## Fetching From Memory



Put the two together



Only the results register has changed: it has disappeared since we will write the result directly to memory. However, this processor is inconvenient:

1. Data that was placed on the data input lines must now be fetched from memory
2. A memory fetch requires three states, so instead of having five states in our controller we need to have 17 (5 + 3 * 4), where 4 is the number of fetches we need to make
3. Data to be written to memory needs an address to go to. How do we get this into the MAR?

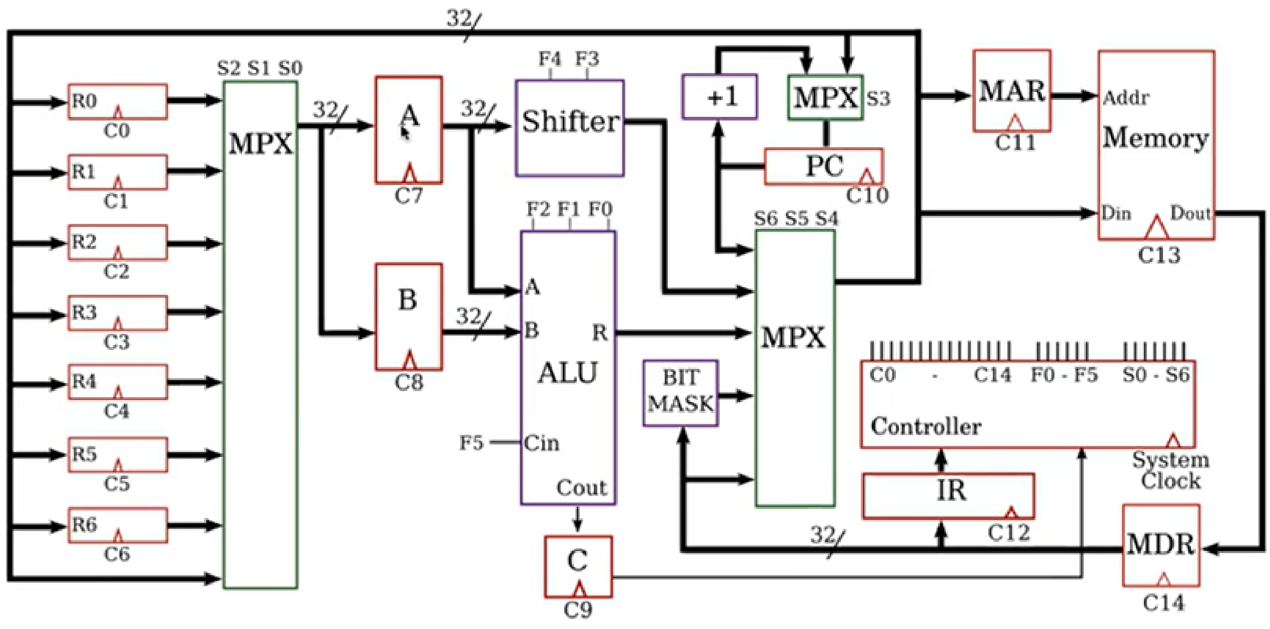
Putting an address in the MAR

1. We could add the address to store the result of a computation as a 5th byte in the program
2. Unfortunately this will mean another Fetch from memory - now we need 20 Fetch/execute states per instruction (5 + 3 * 5)
3. To get the address from the MDR into the MAR will require us to use a path via register A. Unfortunately this will destroy the result current byte stored in A
4. We could feasibly load the MAR before doing the calculation, but this will take another two states - now we need 22 states per instruction (2 + 5 + 3 * 5)
   This is SLOW!

Speed things up

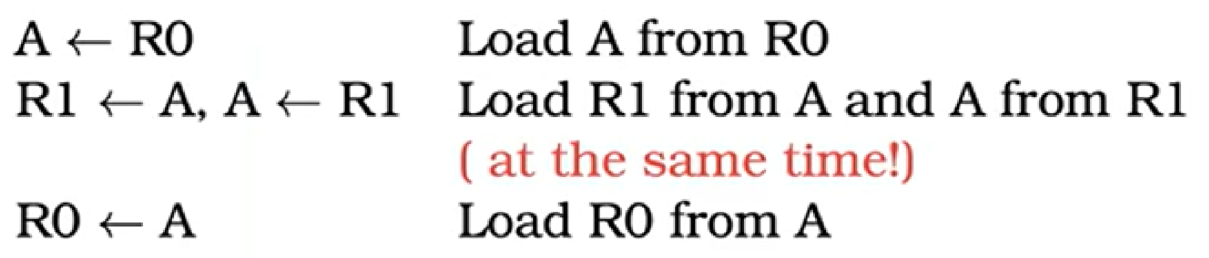
1. Change all buses from 8 bits to 32 bit, thus do four times as much in each cycle
2. Provide local registers which can be programmed to store partial results
3. Design a controller with as small a number of execution cycles as possible
4. Remove elaborate carry arrangements, by doing our arithmetic on big integers (32 bit instead of 8 bit, we don't need the carry arrangements to go from low precision to high precision since we already have high precision)
5. Replace the bi-directional data bus with separate "Data In" and "Data Out" buses for the memory

A real processor



A programmer (or rather, their compiler) can only access the R0-6 registers, keeping the A and B registers safely detached from what can be programmed. Result is still being stored in memory rather than in a Result Register.

Swap values of R0 and R1?



| | |
|---|---|
| A ← R0 | Load A from R0 |
| R1 ← A, A ← R1 | Load R1 from A and A from R1 |
| | ( at the same time!) |
| R0 ← A | Load R0 from A |

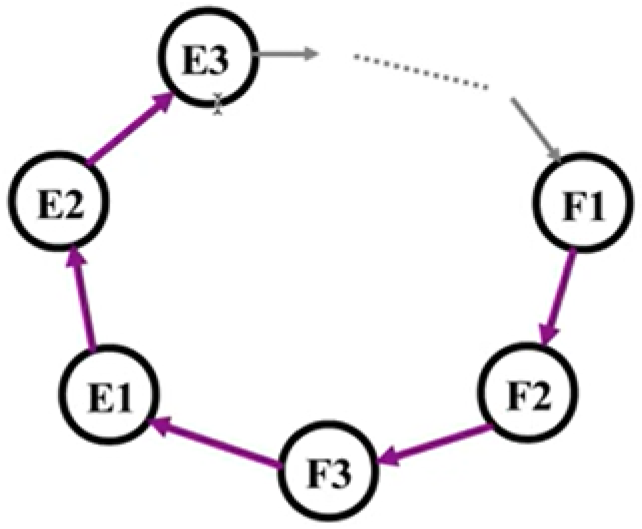
In step 2, two instructions can be executed at the same time because of how DQ flip flops work (data can be loaded into D and taken out through Q at the same time).

Observations

1. Registers A, B, PC, MAR, MDR, IC and C belong to the hardware designers
2. Registers R0-6 can be manipulated by the programmers
3. Most arithmetic operations will no longer require a carry in, so we can set the ALU Cin directly from the controller if required (e.g. for an increment or a jump instruction)

The controller Spec

The controller is going to be a sequential circuit looking something like this:



Always the same three fetch states. During the F states instructions are fetched from memory. There can be any number of Execute states, depending on the complexity of the instruction. During the E states instructions are executed.

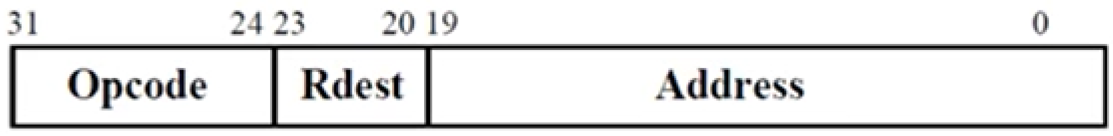
The Fetch cycle will get one 32 bit instruction from the memory. The execute cycles will make the processor carry out that instruction. So, to formalise our spec we need to define what the instructions will do.

The program Instructions
Assembler instructions are a compromise between the software specialists and the hardware designers.
From the software point of view, instructions should be few but very powerful.
From the hardware point of view, instructions should be simple and take only a few clock pulses to execute. A compromise must be found.

## Instruction Format

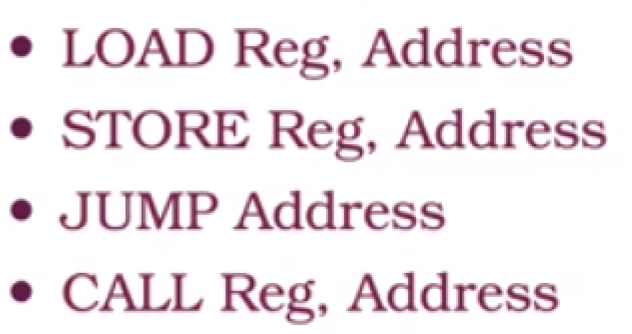
We need to be very precise about our instruction format so that we can easily interpret it in hardware. We will assume that there will be 255 or less instructions, and the top eight bits will define the instruction. Most instructions will act on a register, so we will let the next four bits define the destination register.

We have 7 registers, we only need 3 bits, why do have 4? We reserve one bit to have a little headroom to improve our processor in the next version.



## Memory Reference Instructions

There are four basic instructions that reference the memory directly:

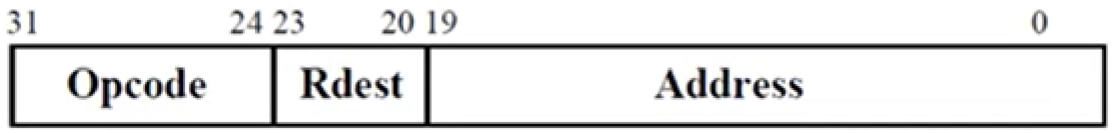


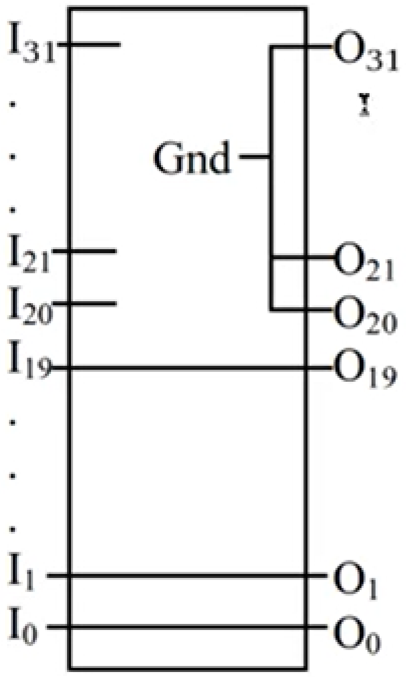
- LOAD Reg, Address
- STORE Reg, Address
- JUMP Address
- CALL Reg, Address

LOAD Reg, Address = Load the contents at Address into Reg
STORE Reg, Address = Store the contents in Reg into address
JUMP Address = Jump to Address (overriding PC)
CALL Reg, Address = Jump to Address, then jump back to return address (stored in Reg)

## Function of bit mask

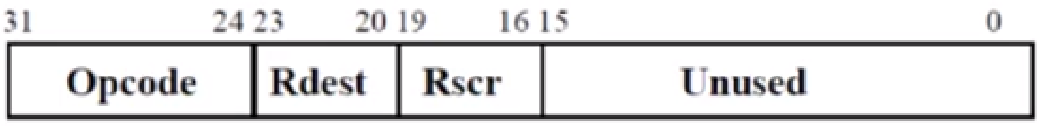
The memory reference instructions contain a memory address in the bottom 20 bits. The bit mask converts this unsigned 20 bit number to a 32 bit number by masking the top bits.





Indirect Memory Reference Instructions
20 bit addresses are a little small - they can only address 1M, so we introduce "indirect addressing" where a 32-bit address is stored in a register. Indirect instructions are slower, but make things easier for the software department.

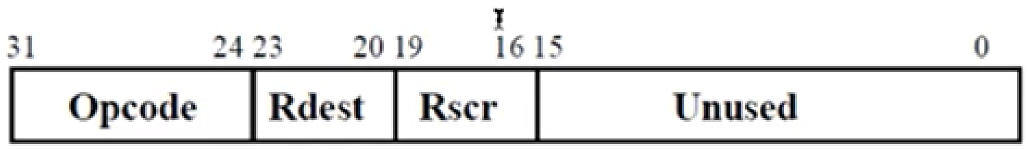
- LOADINDIRECT Reg1, Reg2
- STOREINDIRECT Reg1, Reg2
- JUMPINDIRECT Reg
- CALLINDIRECT Reg1, Reg2



Rscr stores Address.

2 Register Instructions

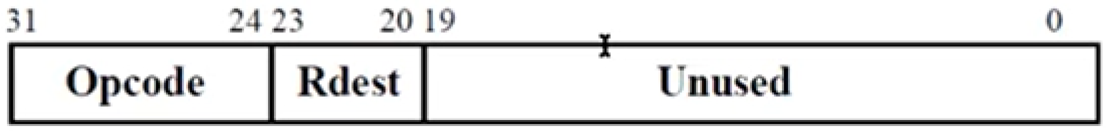
- MOVE Rdest, Rscr
- ADD Rdest, Rscr
- SUBTRACT Rdest, Rscr
- AND Rdest, Rscr
- OR Rdest, Rscr
- XOR Rdest, Rscr
- COMPARE Rdest, Rscr
  (Subtract but without storing the result)



Move: move the content of one register to another.

1 Register Instructions

- CLEAR Rdest
- INCREMENT (INC Rdest)
- DECREMENT (DEC Rdest)
- COMPLEMENT (COMP Rdest)
- ARITHMETIC SHIFT LEFT (ASL Rdest)
- ARITHMETIC SHIFT RIGHT (ASR Rdest)
- ROTATE RIGHT (ROR Rdest)
- RETURN



Complement: Just flipping the bits, not adding one
Return: Used with a CALL instruction, returns to return address

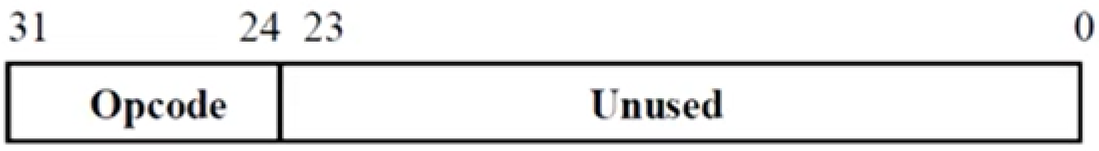
No Register Instructions

Following an arithmetic or compare operation:

1. SKIPPOSITIVE (skip next line if positive)
2. SKIPNEGATIVE (skip next line if negative)

Unconditional

1. SKIP
2. NOP



Input and Output

1. Input and output will be achieved using the LOAD and STORE instructions
2. many peripherals, disks, serial ports, etc can be read or written to as if they were memory
3. They can also be connected directly with the memory so that input and output can be done while the processor is otherwise engaged

Register transfers

1. Each of the instructions that we have specified requires a series of register transfers to carry them out
2. We continue out design, by determining the register transfers that will be need to carry out each instruction
3. These register transfers will become the formal spec of the processor controller

Register Transfers and Processor operation

1. At each step of the execution cycle a number of register transfers take place
2. In the Fetch steps, the register transfers are always the same
3. In the execute steps the register transfers depend on the instruction being executed

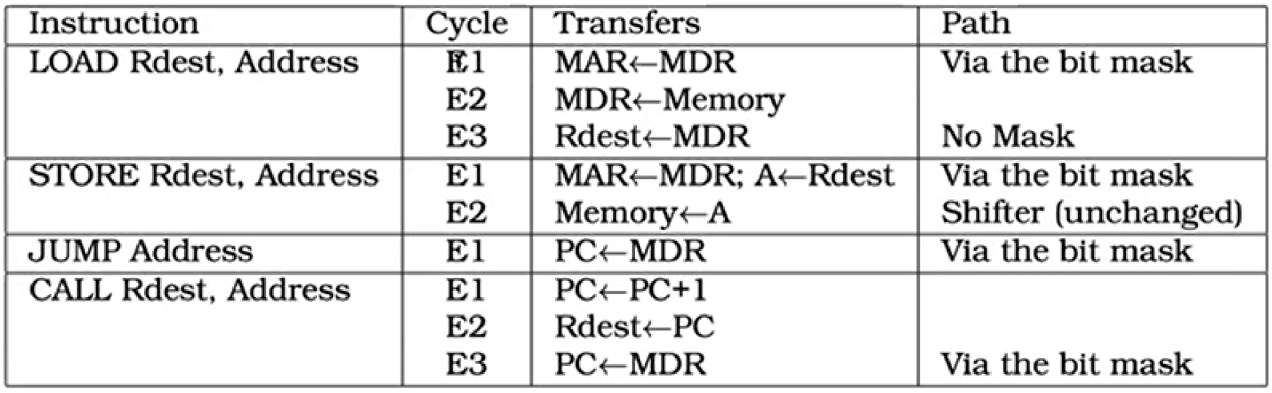
Fetch Register Transfers

The MDR is the only register connected to the memory data out bus (in the version of the processor). Three states are required to fetch an instruction:

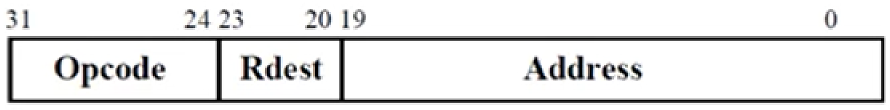
- F1: MAR←PC; PC←PC+1
- F2: MDR←Memory
- F3: IR←MDR

After F3 has finished, the instruction is in both IR and MDR.

Memory reference Instructions

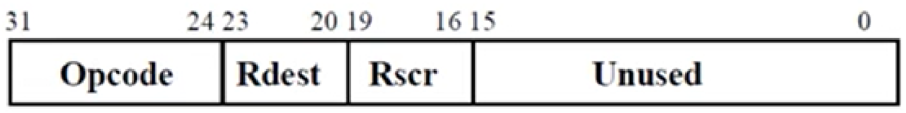


| Instruction | Cycle | Transfers | Path |
|---|---|---|---|
| LOAD Rdest, Address | E1 | MAR←MDR | Via the bit mask |
| | E2 | MDR←Memory | |
| | E3 | Rdest←MDR | No Mask |
| STORE Rdest, Address | E1 | MAR←MDR; A←Rdest | Via the bit mask |
| | E2 | Memory←A | Shifter (unchanged) |
| JUMP Address | E1 | PC←MDR | Via the bit mask |
| CALL Rdest, Address | E1 | PC←PC+1 | |
| | E2 | Rdest←PC | |
| | E3 | PC←MDR | Via the bit mask |



Indirect Memory Reference Instructions

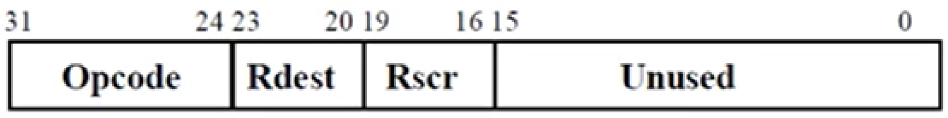
| Instruction | Cycle | Transfers | Path |
|---|---|---|---|
| LOADINDIRCET Rdest, Rscr | E1 | A←Rsrc | |
| | E2 | MAR←A | |
| | E3 | MDR←Memory | |
| | E4 | Rdest←MDR | No Mask |
| STOREINDIRECT Rdest, Rscr | E1 | A←Rscr | |
| | E2 | MAR←A;A←Rdest | via the shifter |
| | E3 | Memory←A | via the shifter |
| JUMPINDIRECT Rscr | E1 | A←Rscr | |
| | E2 | PC←A | via the shifter |
| CALLINDIRECT Rdest, Rscr | E1 | PC←PC+1;A←Rscr | |
| | E2 | Rdest←PC; | |
| | E3 | PC←A | via the shifter |



## 2 Register Instructions

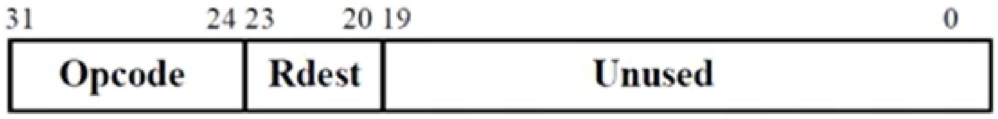
| Instruction | Cycle | Transfers | Path |
|---|---|---|---|
| MOVE Rdest, Rsrc | E1 | A←Rscr | |
| | E2 | Rdest←Shifter | Shifter (unchanged) |
| ADD Rdest, Rsrc | E1 | A←Rscr | |
| | E2 | B←Rdest | |
| | E3 | Rdest←ALUres; C←ALUcout | ALU=A+B, Cin=0 |
| COMPARE Rdest, Rsrc | E1 | A←Rscr | |
| | E2 | B←Rdest | |
| | E3 | C←ALUcout | ALU=A-B, Cin=0 |

(Plus similar arithmetic and logical operations)



## 1 Register Instructions

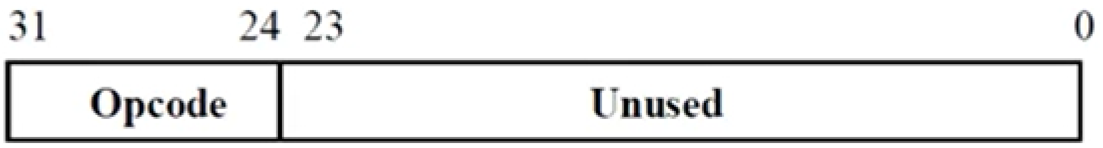
| Instruction | Cycle | Transfers | Path |
|---|---|---|---|
| CLEAR Rdest | E1 | Rdest←ALUres | ALU = zero out |
| INC Rdest | E1<br>E2<br>E3 | A←Rdest<br>B←ALUres<br>Rdest←ALUres;<br>C←ALUcout | <br>ALU = zero out<br>ALU=A+B, Cin=1 |
| DEC Rdest | E1<br>E2<br>E3 | A←Rdest<br>B←ALUres<br>Rdest←ALUres;<br>C←ALUcout | <br>ALU = -1 out<br>ALU=A+B, Cin=0 |
| COMP Rdest | E1<br>E2<br>E3 | A←Rdest<br>B←ALUres<br>Rdest←ALUres | <br>ALU = -1 out<br>ALU=A eor B |
| ASL Rdest | E1<br>E2 | A←Rdest<br>Rdest←Shifter | <br>Shifter (Arithmetic left) |
| RETURN Rdest | E1<br>E2 | A←Rdest<br>PC←Shifter | <br>Shifter (Unchanged) |



No Register Instructions

| Instruction | Cycle | Transfers | Path |
|---|---|---|---|
| SKIP | E1 | PC←PC+1 | |

The conditional skip instructions (SKIPPOSITIVE, SKIPNEGATIVE) also take just one execution cycle and may increment the PC depending on the arithmetic carry. The NOP instruction doesn't need any execute cycles, and has no associated register transfers.


31
24 23
0
Opcode
Unused