# ECE 586 - COMPUTER ARCHITECTURE

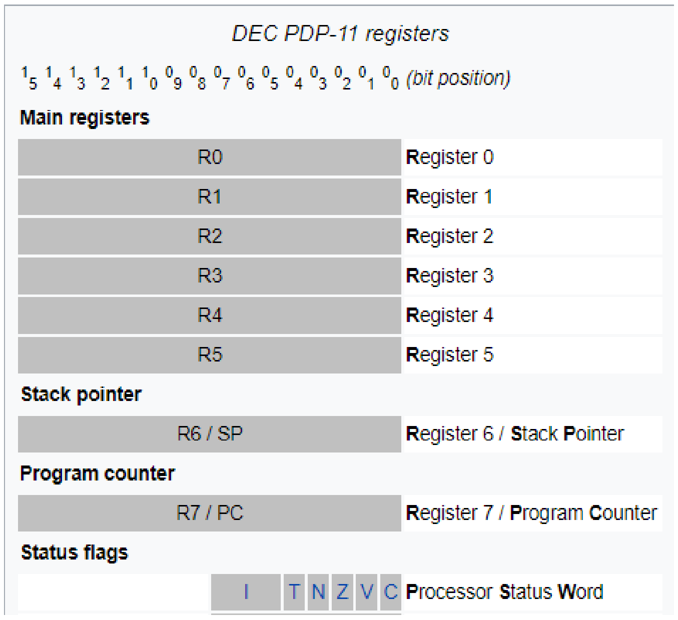
# Final Project Report
# PDP 11 Instruction Set Architecture Simulator

**By**

Abhijit Marathe - amarathe@pdx.edu
Amruta Kalambkar - amruta2@pdx.edu
Manjush Padmamma Venkatesha - manjush@pdx.edu
Vadiraja M N - vadiraja@pdx.edu

## Objective:

**Simulate an instruction set architecture (ISA) for the PDP-11 minicomputer and to generate memory trace files. We are implementing the simulator in Python.**



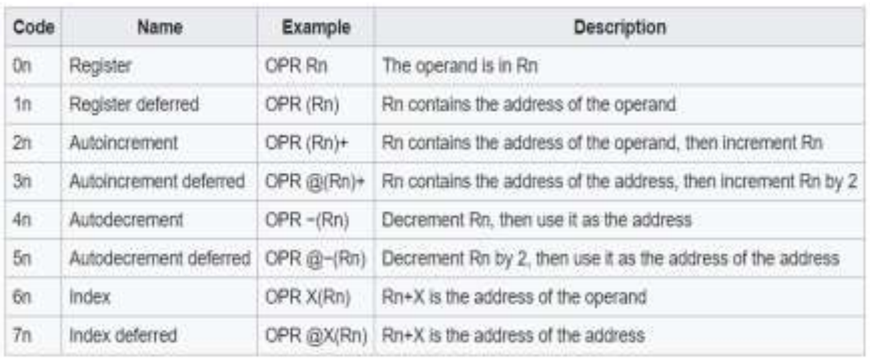
*DEC PDP-11 registers*

15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00 *(bit position)*

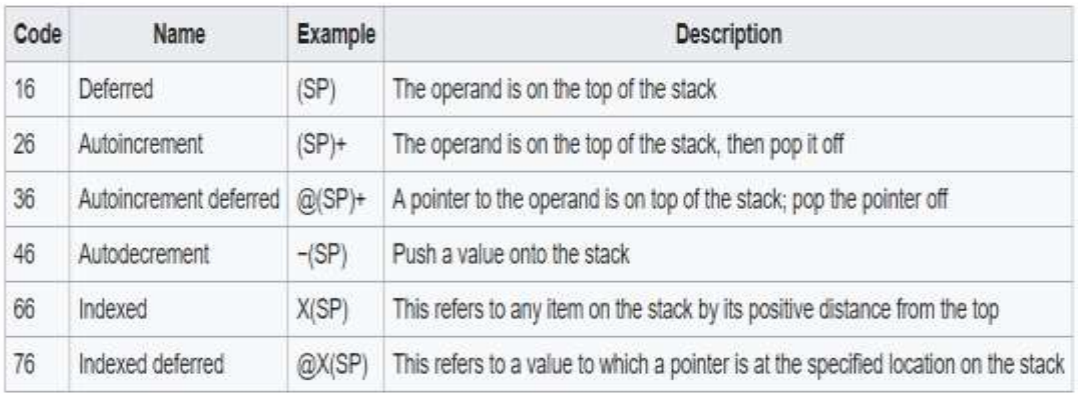
| | |
|---|---|
| **Main registers** | |
| R0 | **R**egister 0 |
| R1 | **R**egister 1 |
| R2 | **R**egister 2 |
| R3 | **R**egister 3 |
| R4 | **R**egister 4 |
| R5 | **R**egister 5 |
| **Stack pointer** | |
| R6 / SP | **R**egister 6 / **S**tack **P**ointer |
| **Program counter** | |
| R7 / PC | **R**egister 7 / **P**rogram **C**ounter |
| **Status flags** | |
| I T N Z V C | **P**rocessor **S**tatus **W**ord |

## General register addressing modes:

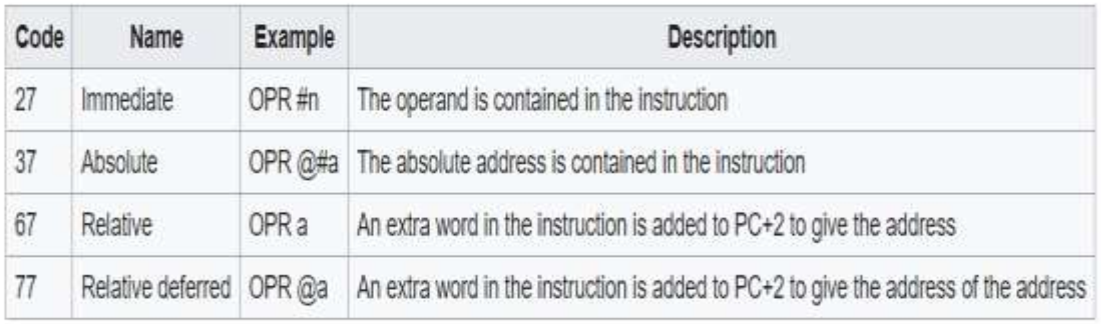
The following eight modes can be applied to any general register. Their effects when applied to R6 (the stack pointer, SP) and R7 (the program counter, PC) are set out separately in the following sections.



| Code | Name | Example | Description |
|---|---|---|---|
| 0n | Register | OPR Rn | The operand is in Rn |
| 1n | Register deferred | OPR (Rn) | Rn contains the address of the operand |
| 2n | Autoincrement | OPR (Rn)+ | Rn contains the address of the operand, then increment Rn |
| 3n | Autoincrement deferred | OPR @(Rn)+ | Rn contains the address of the address, then increment Rn by 2 |
| 4n | Autodecrement | OPR -(Rn) | Decrement Rn, then use it as the address |
| 5n | Autodecrement deferred | OPR @-(Rn) | Decrement Rn by 2, then use it as the address of the address |
| 6n | Index | OPR X(Rn) | Rn+X is the address of the operand |
| 7n | Index deferred | OPR @X(Rn) | Rn+X is the address of the address |



| Code | Name | Example | Description |
|---|---|---|---|
| 16 | Deferred | (SP) | The operand is on the top of the stack |
| 26 | Autoincrement | (SP)+ | The operand is on the top of the stack, then pop it off |
| 36 | Autoincrement deferred | @(SP)+ | A pointer to the operand is on top of the stack; pop the pointer off |
| 46 | Autodecrement | −(SP) | Push a value onto the stack |
| 66 | Indexed | X(SP) | This refers to any item on the stack by its positive distance from the top |
| 76 | Indexed deferred | @X(SP) | This refers to a value to which a pointer is at the specified location on the stack |



| Code | Name | Example | Description |
|---|---|---|---|
| 27 | Immediate | OPR #n | The operand is contained in the instruction |
| 37 | Absolute | OPR @#a | The absolute address is contained in the instruction |
| 67 | Relative | OPR a | An extra word in the instruction is added to PC+2 to give the address |
| 77 | Relative deferred | OPR @a | An extra word in the instruction is added to PC+2 to give the address of the address |

## <u>Design Implementation</u>:

- We stored mnemonic corresponding to the instructions and their opcodes in dictionary ‘Mnemonic’.
- Then we created the function *loadInstr()* to read the input ASCII file and dump it to the memory after which the instruction is loaded from the memory into the instruction register after the completion of the previous instruction.
- Decode(instruction_reg) function decodes the instruction stored in instruction register into 5 types i.e., double operand, single operand, swab, branch or PSW.
- We also define a class *Instruction_Type* which further splits the instruction register into type, opcode, source mode, source register, destination mode, destination register based on whether it is a single operand, double operand or branch instruction.
- Class *Addressing_Modes* defines all the addressing modes and its functionality.
- *Class Operation* checks the mnemonic and calls the corresponding function based on the opcode.

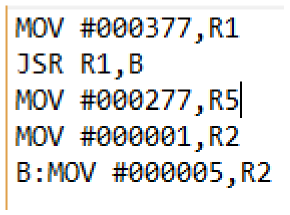
- We then define function for each instruction and its functionality.
- In the main function, one instruction is fetched, decoded and executed by calling functions defined above.
- The function *writeTrace(Operation, Memaddress)* will write to the trace file based on whether it is a data read, data write or an instruction fetch.

## Testing Strategy:

- Check results for all the single and double operands
- Test weather program counter PC is incrementing as expected while performing branch instructions operation results are generating appropriately.
- Test if all the addressing modes are executed as expected.
- Test whether conditional codes are properly set when performing arithmetic operations.
- Expected trace file is being generated.

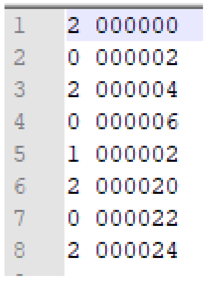
## Results:

Single and Double Operand instructions-



```
MOV #000377,R1
JSR R1,B
MOV #000277,R5
MOV #000001,R2
B:MOV #000005,R2
```

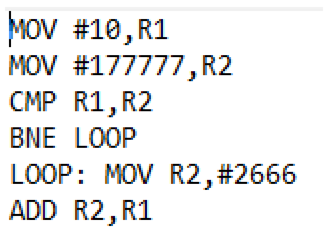
Assembly code



```
2 000000
0 000002
2 000004
0 000006
1 000002
2 000020
0 000022
2 000024
```

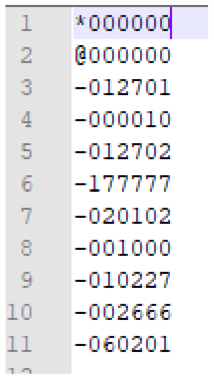
Trace file

Branch instructions-



```
MOV #10,R1
MOV #177777,R2
CMP R1,R2
BNE LOOP
LOOP: MOV R2,#2666
ADD R2,R1
```

Assembly code



```
*000000
@000000
-012701
-000010
-012702
-177777
-020102
-001000
-010227
-002666
-060201
```

Trace file

## <u>**References**</u>:

- https://en.wikipedia.org/wiki/PDP-11_architecture
- http://gordonbell.azurewebsites.net/digital/pdp%2011%20handbook%201969.pdf
- http://pages.cpsc.ucalgary.ca/~dsb/PDP11/