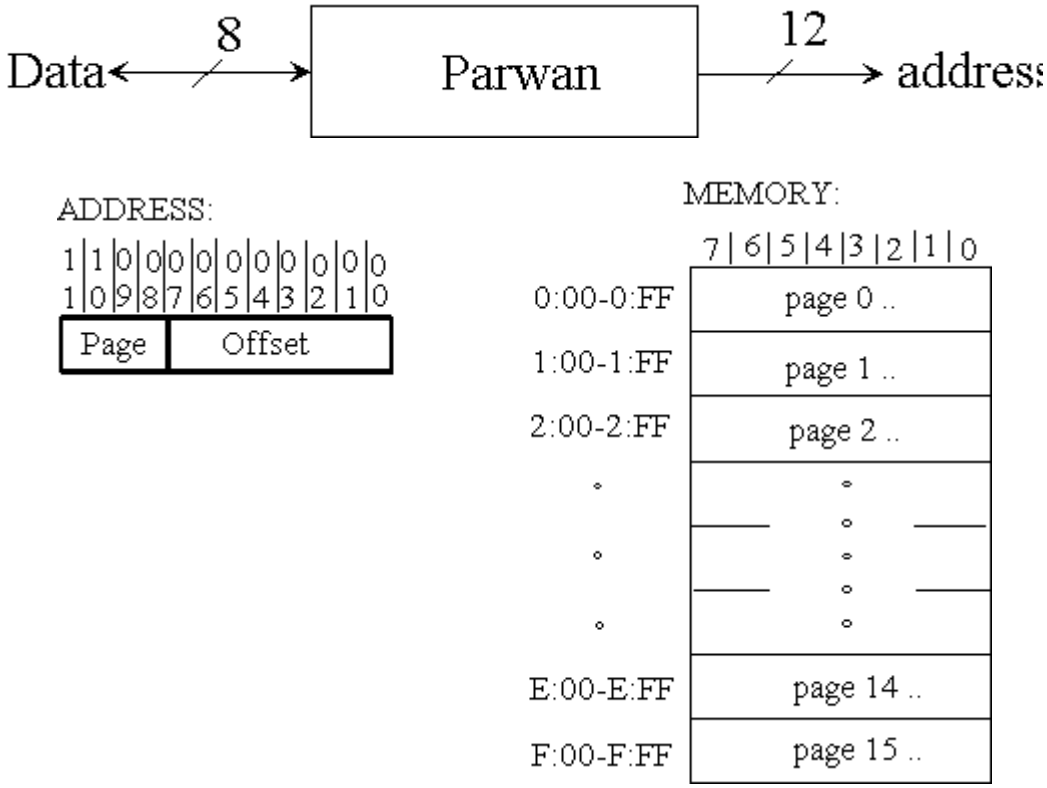
PART 4

General CPU description

General CPU description

- **PARWAN; *PAR_1*; A Reduced Processor**
- **Simple 8-bit CPU**
- **8-bit Data; 12-bit Address**
- **Primarily designed for educational purposes**
- **Includes most common instructions**

General CPU description



- **12-Bit Address Bus (4-Bit Page Address + 8-Bit Offset Within Page)**
- **Memory divided into pages (2^4 = 16 Page)**
- **Pages of 2^8 = 256 bytes**
- **12-Bit Address (3 Hex Numbers) has page and offset part { X : YZ }**

(X = Page Number, YZ = Offeset Within Page)

- 8-Bit Data Bus ===>> 16 Page of 256 BYTES

- Uses memory mapped IO

Instruction Set

1. FULL Address Instructions ==>> (12 bits & Direct / Indirect)

   LDA, AND, ADD, SUB, JMP, STA

2. PAGE Address Instructions ==>> (8 bit & Only Direct Addressing)

   JSR, BRA_V, BRA_C,
   BRA_Z, BRA_N

3. NO Address Instructions

NOP, CLA, CMA, CMC,
ASL, ASR

- **Full Address Instructions include page and offset**
  ==>> 2-Byte Instructions
- **Page address instructions include offset**
  ==>> 2-Byte Instructions
- **No Address instructions occupy a single byte**
  ==>> 1-Byte Instructions

Instruction Set Description.

Instruction Mnemonic	Brief Description
LDA loc	Load AC w/(contents of loc)
A ND loc	AND AC w/(contents of loc)
ADD loc	Add (contents of loc) to AC
SUB loc	Sub (contents of loc) from AC
JMB adr	Jump to adr
STA loc	Store AC in contents of loc
JSR tos	Subroutine to tos
BRA-V adr	Branch to adr if V
BRA-C adr	Branch to adr if C
BRA-Z adr	Branch to adr if Z
BRA-N adr	Branch to adr if N
NOP	No operation
CLA	Clear AC
CMA	Complement AC
CMC	Complement carry
ASL	Arith shift left
ASR	Arith shift right

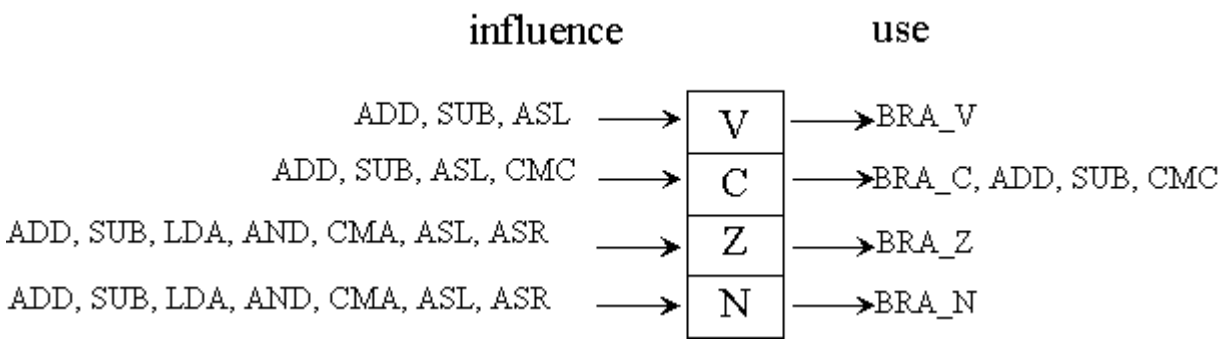
- **Load and store operations**
- **Arithmetic & logical operations**
- ***jmp* and *branch* instructions**

Instruction Set Description.

Instruction Mnemonic	Brief Description	Bits	ADDRESSING Scheme	Indirect	FLAGS use	set	
LDA loc	Load AC w/(contents of loc)	12	FULL	YES	----	--zn	
A ND loc	AND AC w/(contents of loc)	12	FULL	YES	----	--zn	
ADD loc	Add (contents of loc) to AC	12	FULL	YES	-c--	vczn	
SUB loc	Sub (contents of loc) from AC	12	FULL	YES	-c--	vczn	
JMB adr	Jump to adr	12	FULL	YES	----	----	
STA loc	Store AC in contents of loc	12	FULL	YES	----	----	
JSR tos	Subroutine to tos	8	PAGE	NO	----	----	
BRA-V adr	Branch to adr if V	8	PAGE	NO	v---	----	
BRA-C adr	Branch to adr if C	8	PAGE	NO	-c--	----	
BRA-Z adr	Branch to adr if Z	8	PAGE	NO	--z-	----	
BRA-N adr	Branch to adr if N	8	PAGE	NO	---n	----	
NOP	No operation	-	NONE	NO	----	----	
CLA	Clear AC	-	NONE	NO	----	----	
CMA	Complement AC	-	NONE	NO	----	--zn	
CMC	Complement carry	-	NONE	NO	-c--	-c--	
ASL	Arith shift left	-	NONE	NO	----	vczn	
ASR	Arith shift right	-	NONE	NO	----	--zn	

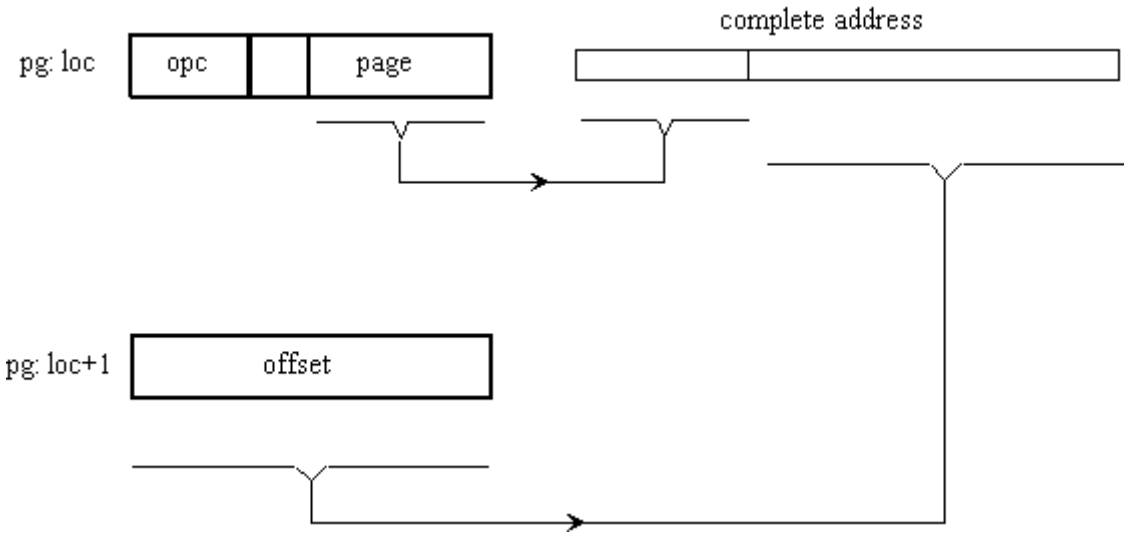
- **CPU contains *V C Z N* flags**
- **Instructions use and/or influence these flags**

Status Register (Flags)



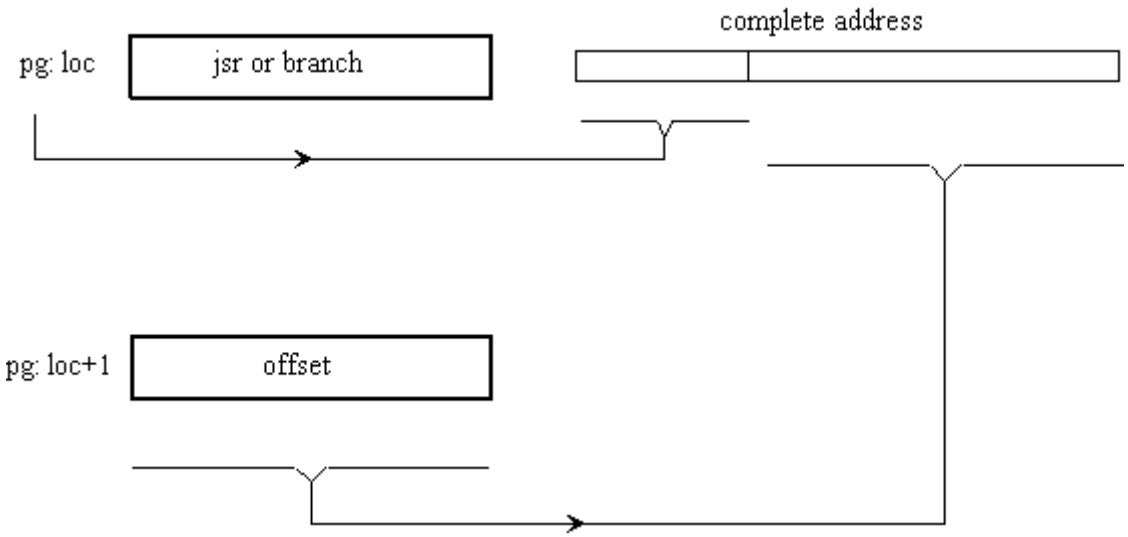
- **Arithmetic instructions influence all flags**
- **Branch instructions use corresponding flags**
- **Shift instructions influence all flags**

Full Addressing



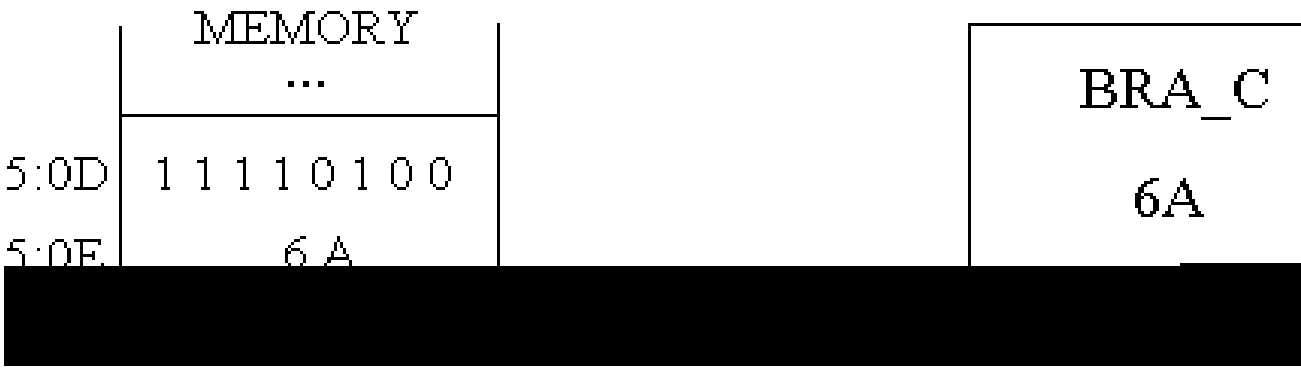
- **Full address instructions use two bytes**
- **Right hand side of first byte is Page #**
- **Second byte contains offset**
- **Bit 4 is Direct / Indirect indicator**

Page Addressing



- **Page address instructions use two bytes**
- **All of first byte is used by opcode**
- **Page part of address uses current page**
- **Second byte is the offset**

Addressing



BRANCH TO 6A if Carry is set Else GoTo 5:0F

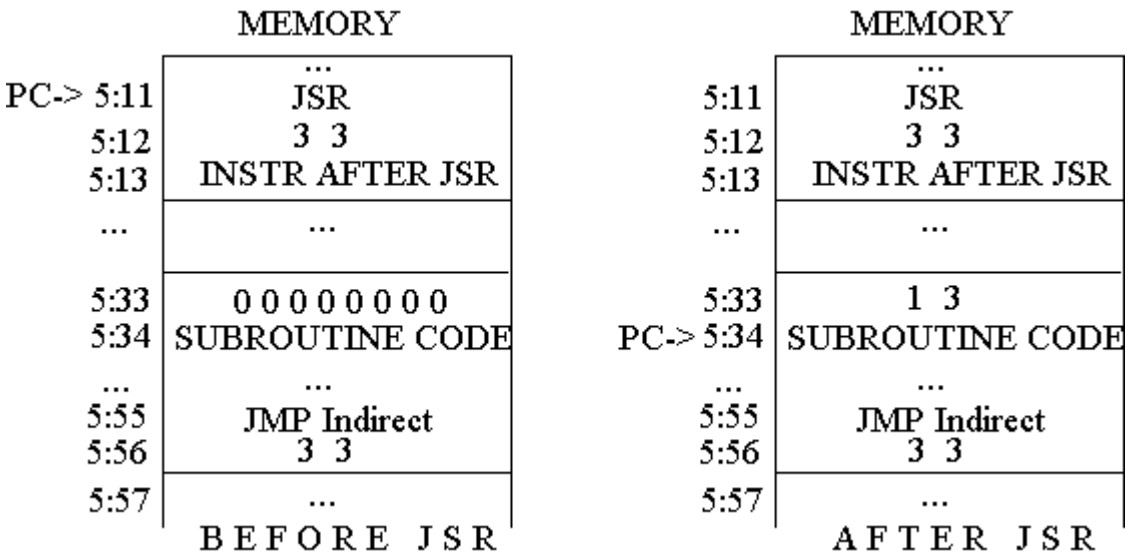
OpCode (Branch if C=1) = 1111_0100

c=0 : Next instruction from 5:0f

c=1 : Next instruction from 5:6A

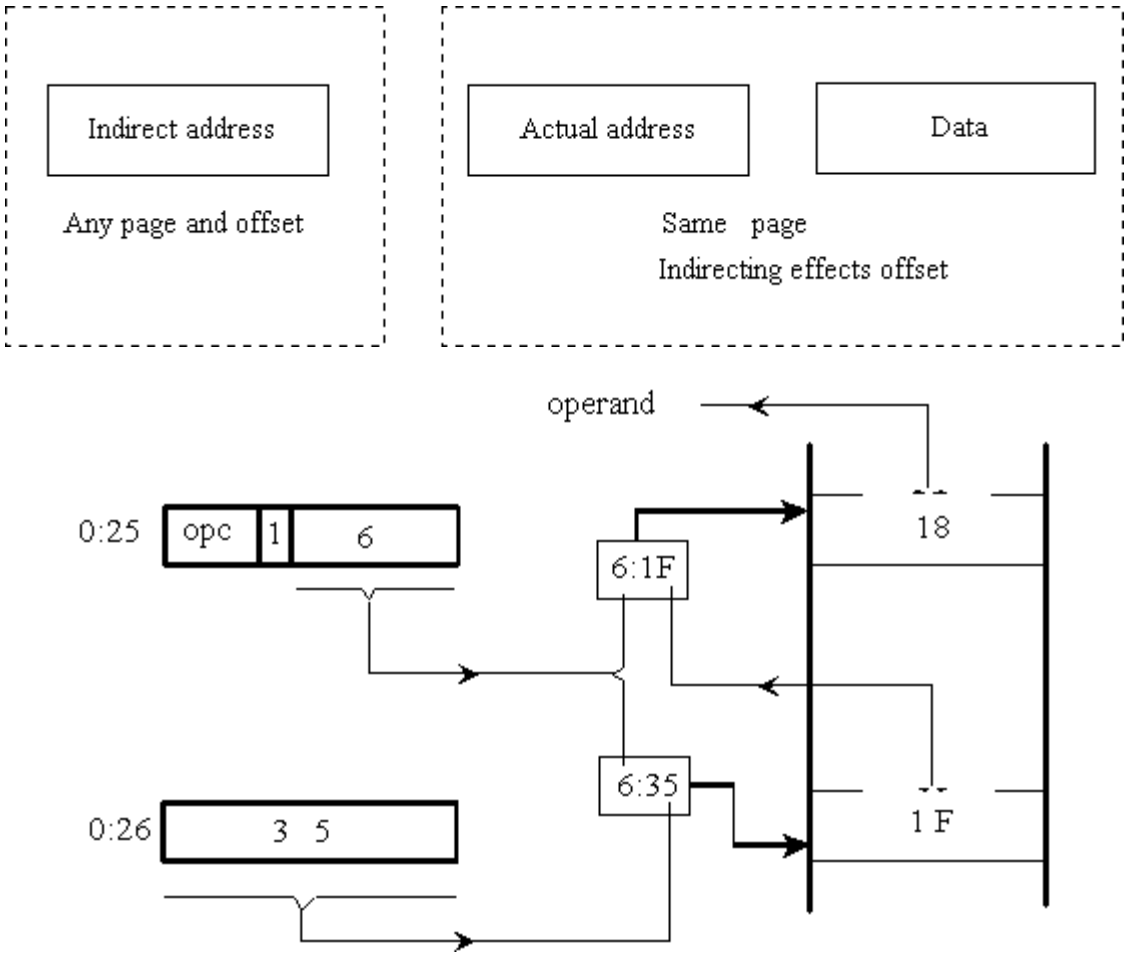
- **Branching is done within current page only**

Addressing (JSR)



- **Store *jsr* return address at *tos***
- **Begin subroutine at *tos*+1**
- **Use indirect *jmp* to *tos* for return from subroutine**

Indirect Addressing



- **Indirect addressing effects offset only**

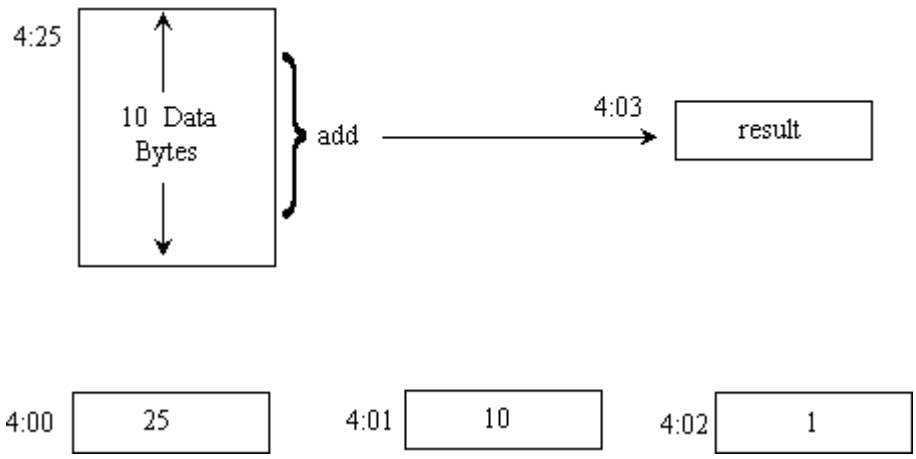
- **Indirect Address = 6:35 ==>> This Memory Byte Contains 1F**

- **Actual Address Accessed ==>> 6:1F**

Instruction Set & Opcodes

Instruction Mnemonic	Opcode Bits 7 6 5	D/I Bit 4	Bits 3 2 1 0
LDA loc	0 0 0	0/1	Page adr
AND loc	0 0 1	0/1	Page adr
ADD loc	0 1 0	0/1	Page adr
SUB loc	0 1 1	0/1	Page adr
JMP adr	1 0 0	0/1	Page adr
STA loc	1 0 1	0/1	Page adr
JSR tos	1 1 0	-	- - - -
BRA_V adr	1 1 1	1	1 0 0 0
BRA_C adr	1 1 1	1	0 1 0 0
BRA_Z adr	1 1 1	1	0 0 1 0
BRA_N adr	1 1 1	1	0 0 0 1
NOP	1 1 1	0	0 0 0 0
CLA	1 1 1	0	0 0 0 1
CMA	1 1 1	0	0 0 1 0
CMC	1 1 1	0	0 1 0 0
ASL	1 1 1	0	1 0 0 0
ASR	1 1 1	0	1 0 0 1

Code Example (Program to Add 10 #'s==> Location 4:00 Points To Data Use Location 4:01 To Store Data Count -- Constant 1 Stored in 4:02 for +1 & -1)



0:15 cla -- clear accumulator
0:16 asl -- clears carry

0:17 add, i 4:00 -- add bytes (Partial Sum Updated)
0:19 sta 4:03 -- Store Updated Partial Sum

0:1B lda 4:00 -- load pointer
0:1D add 4:02 -- increment pointer (Update Pointer To Point

0:1F sta 4:00 -- store pointer back To the Next

0:21 lda 4:01 -- load count
0:23 sub 4:02 -- decrement count (Decrement Counter

0:25 bra_z :2D -- end if zero count Exit if Counter=0)
0:27 sta 4:01 -- store count back

0:29 lda 4:03 -- get partial sum
0:2B jmp 0:17 -- go for next byte

0:2D nop -- adding completed

General CPU Description

Uilities

Basic Utilities (Developed Before)

```
PACKAGE basic_utilities IS
TYPE integers IS ARRAY (0 TO 12) OF INTEGER;
FUNCTION fgl (w, x, gl : BIT) RETURN BIT;
FUNCTION feq (w, x, eq : BIT) RETURN BIT;
PROCEDURE bin2int (bin : IN BIT_VECTOR; int :
OUT INTEGER);
PROCEDURE int2bin (int : IN INTEGER; bin : OUT
BIT_VECTOR);
PROCEDURE apply_data (SIGNAL target : OUT
BIT_VECTOR (3 DOWNTO 0);
CONSTANT values : IN integers; CONSTANT period :
IN TIME);
END basic_utilities;
```

Par- Utilities

```
LIBRARY cmos;
USE cmos.basic_utilities.ALL;
--
```

```
PACKAGE par_utilities IS
FUNCTION "XOR" (a, b : qit) RETURN qit ;
--
FUNCTION "AND" (a, b : qit_vector) RETURN
qit_vector;

FUNCTION "OR" (a, b : qit_vector) RETURN
qit_vector;
FUNCTION "NOT" (a : qit_vector) RETURN
qit_vector;
--
SUBTYPE nibble IS qit_vector (3 DOWNTO 0);
SUBTYPE byte IS qit_vector (7 DOWNTO 0);
SUBTYPE twelve IS qit_vector (11 DOWNTO 0);
--
SUBTYPE wired_nibble IS wired_qit_vector (3
DOWNTO 0);
SUBTYPE wired_byte IS wired_qit_vector (7
DOWNTO 0);
SUBTYPE wired_twelve IS wired_qit_vector (11
DOWNTO 0);
--
SUBTYPE ored_nibble IS ored_qit_vector (3
DOWNTO 0);
SUBTYPE ored_byte IS ored_qit_vector (7
DOWNTO 0);
SUBTYPE ored_twelve IS ored_qit_vector (11
DOWNTO 0);
--
CONSTANT zero_4 : nibble := "0000";
CONSTANT zero_8 : byte := "00000000";
```

```
CONSTANT zero_12 : twelve := "000000000000";
--
FUNCTION add_cv (a, b : qit_vector; cin : qit)
RETURN qit_vector;
FUNCTION sub_cv (a, b : qit_vector; cin : qit)
RETURN qit_vector;
--
FUNCTION set_if_zero (a : qit_vector) RETURN
qit;
--
END par_utilities;
```

```
PACKAGE body par_utilities IS


FUNCTION "XOR" (a, b : qit) RETURN qit
IS
CONSTANT qit_or_table : qit_2d :=
(

('0','1','1','X'),

('1','0','0','X'),

('1','0','0','X'),

('X','X','X','X'));
BEGIN
RETURN qit_or_table (a, b);
END "XOR";
```

```
FUNCTION "AND" (a,b : qit_vector) RETURN
qit_vector IS
VARIABLE r : qit_vector (a'RANGE);
BEGIN
   loop1: FOR i IN a'RANGE LOOP
          r(i) := a(i) AND b(i);
   END LOOP loop1;
RETURN r;
END "AND";
--
FUNCTION "OR" (a,b: qit_vector) RETURN
qit_vector IS
VARIABLE r: qit_vector (a'RANGE);
BEGIN
   loop1: FOR i IN a'RANGE LOOP
      r(i) := a(i) OR b(i);
   END LOOP loop1;
RETURN r;
END "OR";
--
FUNCTION "NOT" (a: qit_vector) RETURN
qit_vector IS
VARIABLE r: qit_vector (a'RANGE);
BEGIN
      loop1: FOR i IN a'RANGE LOOP
         r(i) := NOT a(i);
      END LOOP loop1;
RETURN r;
END "NOT";
```

```
FUNCTION add_cv (a, b : qit_vector; cin : qit)
RETURN qit_vector IS
--left bits are sign bit
VARIABLE r, c: qit_vector (a'LEFT + 2
DOWNTO 0);
-- two extra bits in r are: msb for overflow, next
carry
VARIABLE a_sign, b_sign: qit;
BEGIN
a_sign := a(a'LEFT); b_sign := b(b'LEFT);
r(0) := a(0) XOR b(0) XOR cin;
c(0) := ((a(0) XOR b(0)) AND cin) OR (a(0) AND
b(0));
FOR i IN 1 TO (a'LEFT) LOOP
r(i) := a(i) XOR b(i) XOR c(i-1);
c(i) := ((a(i) XOR b(i)) AND c(i-1)) OR (a(i) AND
b(i));
END LOOP;
r(a'LEFT+1) := c(a'LEFT);
IF a_sign = b_sign AND r(a'LEFT) /= a_sign
THEN r(a'LEFT+2) := '1'; --overflow
ELSE r(a'LEFT+2) := '0'; END IF;

RETURN r;
END add_cv;
```

```
FUNCTION sub_cv (a, b : qit_vector; cin : qit)
RETURN qit_vector IS
VARIABLE not_b : qit_vector (b'LEFT
DOWNTO 0);
VARIABLE not_c : qit;
VARIABLE r : qit_vector (a'LEFT + 2 DOWNTO
0);
BEGIN
not_b := NOT b; not_c := NOT cin;
r := add_cv (a, not_b, not_c);

RETURN r;
END sub_cv;
```

```
FUNCTION set_if_zero (a : qit_vector) RETURN
qit IS
VARIABLE zero : qit := '1';
BEGIN
FOR i IN a'RANGE LOOP
IF a(i) /= '0' THEN zero := '0'; EXIT;
END IF;
END LOOP; RETURN zero;
END set_if_zero;


END par_utilities;
```

- ***add_cv*** **adds its operands creates *c* and *v* bits ==>> r = a + b + Cin**
- **Put overflow in leftmost result bit**
- **Put carry to the right of overflow**

- **Sub_Cv performs the Subtraction ==>> r = a - (b + Cin)**

Op-Code Definitions

```
LIBRARY cmos;
USE cmos.basic_utilities.ALL;
--
PACKAGE par_parameters IS
CONSTANT single_byte_instructions : qit_vector (3 DOWNTO 0) := "1110";
CONSTANT cla : qit_vector (3 DOWNTO 0) := "0001";
CONSTANT cma : qit_vector (3 DOWNTO 0) := "0010";
CONSTANT cmc : qit_vector (3 DOWNTO 0) := "0100";
CONSTANT asl : qit_vector (3 DOWNTO 0) := "1000";
CONSTANT asr : qit_vector (3 DOWNTO 0) := "1001";
CONSTANT jsr : qit_vector (2 DOWNTO 0) := "110";
CONSTANT bra : qit_vector (3 DOWNTO 0) := "1111";
```

```
CONSTANT indirect : qit :=
'1';
CONSTANT jmp : qit_vector (2 DOWNTO 0) :=
"100";
CONSTANT sta : qit_vector (2 DOWNTO 0) :=
"101";
CONSTANT lda : qit_vector (2 DOWNTO 0) :=
"000";
CONSTANT ann : qit_vector (2 DOWNTO 0) :=
"001";
CONSTANT add : qit_vector (2 DOWNTO 0) :=
"010";
CONSTANT sbb : qit_vector (2 DOWNTO 0) :=
"011";
END par_parameters;
```

- **Assign appropriate names to opcodes**
- ***par_parameters* is used for readability**

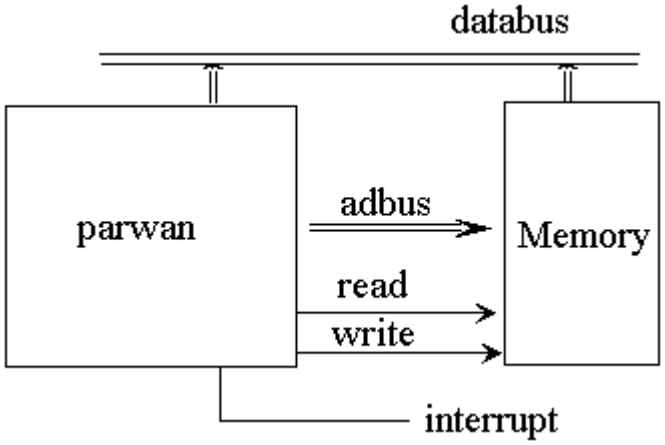
General CPU Description

1. **Libraries**

2. **Interface**

3. **Architecture**

Interface Description



- *Use a Single Process to Describe General Machine Operation*

```
LIBRARY cmos;
USE cmos.basic_utilities.ALL;
LIBRARY par_library;
USE par_library.par_utilities.ALL;
USE par_library.par_parameters.ALL;
--
ENTITY par_central_processing_unit IS
GENERIC (read_high_time, read_low_time,
                write_high_time, write_low_time :
TIME := 2 US;
                cycle_time : TIME := 4 US; run_time :
TIME := 140 US);
PORT (clk : IN qit;
          interrupt : IN qit;
          read_mem, write_mem : OUT qit;
```

```
        databus : INOUT wired_byte BUS :=
"ZZZZZZZZ";
        adbus : OUT twelve
    );
END par_central_processing_unit;
```

- Make packages visible
- Databus can be driven by Parwan and Memory
- Use wiring resolution function
- Generic parameters specify relative read/write cycle time
-
- Pseudo Code is First Used To Behaviorally Describe Architecture

---

```
ARCHITECTURE behavioral OF
par_central_processing_unit IS
BEGIN
  PROCESS
    Declare necessary variables;
  BEGIN
      IF NOW > run_time THEN WAIT; END
IF;
      IF interrupt = '1' THEN
            Handle interrupt;
      ELSE -- no interrupt
            Read first byte into byte1, increment
pc; -- Instruction Fetch
            IF byte1 (7 DOWNTO 4) =
```

```
single_byte_instructions THEN
                        Execute single_byte
instructions;
                ELSE -- two-byte
instructions

                        Read second byte into byte2,
increment pc;---- 2nd Byte Fetch
                IF byte1 (7 DOWNTO 5) = jsr THEN
                        Execute jsr instruction; --
byte2 has offset address
                ELSIF byte1 (7 DOWNTO 4) = bra
THEN
                        Execute bra instructions; --
byte2 has offset address
                ELSE -- all other two-byte
instructions
                    IF byte1 (4) = indirect THEN  --
Indirect Bit is Set
                        Use byte1 and byte2 to get
address;
                    END IF; -- ends
indirect
                IF byte1 (7 DOWNTO 5) = jmp
THEN
                        Execute jmp instruction,
                ELSIF byte1 (7 DOWNTO 5) = sta
THEN
                        Execute sta instruction, write
ac;
```

```
                    ELSE -- read operand for lda, and, add, sub
                        Read memory onto databus;
                        Execute lda, and, add, and sub;
                        Disconnect memory from databus;
                    END IF; -- jmp / sta / lda, and, add, sub
                    END IF; -- jsr / bra / other double-byte instructions
                    END IF; -- single-byte / double-byte
END IF; -- interrupt / otherwise
END PROCESS;
END behavioral;
```

Coding of Individual Instructions

```
Declare necessary variables
VARIABLE pc : twelve;
VARIABLE ac, byte1, byte2 : byte;
VARIABLE v, c, z, n : qit;
VARIABLE temp : qit_vector (9 DOWNTO 0);
```

Handle interrupt

```
pc := zero_12; -- Interrupt Handling Routine is Located at Memory Address 0
WAIT FOR cycle_time;
Read first byte into byte1, increment pc
```

```
adbus <= pc;                -- Start A Memory Read
Cycle
read_mem <=
'1';
WAIT FOR read_high_time; -- Memory Access
Delay
byte1 := byte (databus); -- databus is Type-Cast to
byte
read_mem <=
'0';
WAIT FOR read_low_time; -- Prevents
OverWriting
pc := inc (pc);
```

Memory READ CYCLE

1. Put *address* on address bus
2. Wait half a clock cycle
3. Read data bus
4. Remove read request

Execute single_byte instructions

```
CASE byte1 (3 DOWNTO 0) IS
WHEN cla =>   ac := zero_8;  z := '1';
WHEN cma => ac := NOT ac;
                    IF ac = zero_8 THEN z := '1';
END IF;
                    n := ac (7);
WHEN cmc =>  c := NOT c;
```

```
WHEN asl =>    c := ac (7);
                ac := ac := ac (6 DOWNTO 0) &
'0';
               n := ac (7);
               IF c /= n THEN v := '1'; END IF;
WHEN asr =>    ac := ac (7) & ac (7 DOWNTO 1);
               IF ac = zero_8 THEN z := '1';
END IF;
               n := ac (7);
WHEN OTHERS => NULL;
END CASE;
```

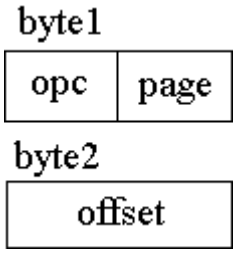
- Handing *single_byte* instructions, *cla*, *cma*, *cmc*, *asl* and *asr*
- Negative flag may be set for *cma*, *asl* and *asr*
- Zero flag may be set for *cma*, *asl* and *asr*
- For *asl*, overflow occurs it bits 6 & 7 differ

Execute Two byte instructions

Read second byte into *byte2*, increment pc

```
adbus <= pc;
read_mem <= '1';
WAIT FOR read_high_time;
byte2 := byte (databus);
read_mem <= '0';
WAIT FOR read_low_time;
```

pc := inc (pc);



- **Reading byte from memory**
- ***Read_memory* stays high for half a clock**
- **Memory releases the bus in the second half**
- **Right half of *byte1* has page for full address**
- ***Byte2* now has the offset of address**

Execute *jsr* instruction, *byte2* has address

databus <= wired_byte (pc (7 DOWNTO 0)); *-- Offset Part of PC is*

--

Written to DATA Bus

adbus (7 DOWNTO 0) <= byte2; *-- Offset Part of Subroutine Address is*

-- Placed on Address Bus (Page Address already there

Write_mem <= '1';

WAIT FOR write_high_time;
write_mem <= '0';

```
WAIT FOR write_low_time;
databus <= "ZZZZZZZZ";
pc (7 DOWNTO 0) := inc (byte2);
```

- Handling *jsr*
- Page part of *adbus* still points to the same instruction page
- Write *pc* to page location pointed by *byte2*
- when writing is done, release the *databus*
- Load *pc* to start from *byte2*+1 (*tos*)

Coding_individual_instructions

```
Execute bra instructions, address in byte2
IF
( byte1 (3) = '1' AND v = '1' ) OR
( byte1 (2) = '1' AND c = '1' ) OR
```

```
( byte1 (1) = '1' AND z = '1' ) OR
( byte1 (0) = '1' AND n = '1' )
THEN
pc (7 DOWNTO 0) := byte2;
END IF;
```

- Chock bits 3, 2, 1 and 0 against *v*, *c*, *z*, *n* flags
- Load *pc* with *byte2* if match is found
- Page part of *pc* still holds some page

Coding individual instructions

```
Use byte1 and byte2 to get address
adbus (11 DOWNTO 8) <= byte1 (3 DOWNTO 0);
adbus (7 DOWNTO 0) <= byte2;
read_mem <= '1'; WAIT FOR read_high_time;
byte2 := byte (databus);
read_mem <= '0'; WAIT FOR read_low_time;
```

- **Use page of *byte1*, offset of *byte2***
- **Form an address to fetch offset of operand**
- **Now *byte1* & *byte2* contain full operand address**

Coding individual instructions

Execute *jmp* instruction
pc := byte1 (3 DOWNTO 0) & byte2;

- **Load *pc* with full 12-bit address**
- **Could use two assignments instead of &**

Coding individual instructions

Execute *sta* instruction, write *ac*
adbus <= byte1 (3 DOWNTO 0) & byte2;
databus <= wired_byte (ac);
write_mem <= '1'; WAIT FOR write_high_time;
write_mem <= '0'; WAIT FOR write_low_time;
databus <= "ZZZZZZZZ";

- **Put full address on *adbus***
- **Put *ac* on *databus***
- **Issue write, when done, release *databus***

Coding individual instructions

Read memory onto *databus*
adbus (11 DOWNTO 8) <= byte1 (3 DOWNTO 0);
adbus (7 DOWNTO 0) <= byte2;
read_mem <= '1'; WAIT FOR read_high_time;
CASE byte1 (7 DOWNTO 5) IS
WHEN lda =>
ac := byte (databus);
WHEN ann =>
ac := ac AND byte (databus);
WHEN add =>
temp := add_cv (ac, byte (databus), c);
ac := temp (7 DOWNTO 0);
c := temp (8);
v := temp (9);

```
WHEN sbb =>
temp := sub_cv (ac, byte (databus), c);
ac := temp (7 DOWNTO 0);
c := temp (8);
v := temp (9);
WHEN OTHERS => NULL;
END CASE;
IF ac = zero_8 THEN z := '1'; END IF;
n := ac (7);
read_mem <= '0'; WAIT FOR read_low_time;
```

- **Full address on adbus**
- **Issue *read_mem***
- **Perform *lda*, *ann*, *add* and *sbb***
- **Arithmetic operations set *c* and *v* flags**
- **All operations set *z* and *n* flags**

Complete Behavioral (Pt 1)

```
 ARCHITECTURE behavioral OF par_central_processing_unit IS
BEGIN
PROCESS
VARIABLE pc : twelve;
VARIABLE ac, byte1, byte2 : byte;
VARIABLE v, c, z, n : qit;
VARIABLE temp : qit_vector (9 DOWNTO 0);
VARIABLE pc : twelve;
VARIABLE ac, byte1, byte2 : byte;
VARIABLE v, c, z, n : qit;
VARIABLE temp : qit_vector (9 DOWNTO 0);
BEGIN
IF NOW > run_time THEN WAIT; END IF;
IF interrupt = '1' THEN
pc := zero_12;
WAIT FOR cycle_time;
```

```
ELSE -- no interrupt
adbus <= pc;
read_mem <= '1'; WAIT FOR read_high_time;
byte1 := byte (databus);
read_mem <= '0'; WAIT FOR read_low_time;
pc := inc (pc);
IF byte1 (7 DOWNTO 4) = single_byte_instructions THEN
CASE byte1 (3 DOWNTO 0) IS
WHEN cla =>
ac := zero_8;
WHEN cma =>
ac := NOT ac;
IF ac = zero_8 THEN z := '1'; END IF;
n := ac (7);
WHEN cmc =>
c := NOT c;
WHEN asl =>
c := ac (7);
ac := ac (6 DOWNTO 0) & '0';
n := ac (7);
IF c /= n THEN v := '1'; END IF;
WHEN asr =>
ac := ac (7) & ac (7 DOWNTO 1);
IF ac = zero_8 THEN z := '1'; END IF;
n := ac (7);
WHEN OTHERS => NULL;
END CASE;
ELSE -- two-byte instructions
adbus <= pc;
read_mem <= '1'; WAIT FOR read_high_time;
byte2 := byte (databus);
read_mem <= '0'; WAIT FOR read_low_time;
pc := inc (pc);
```

Complete Behavioral (Pt 2)

```
IF byte1 (7 DOWNTO 5) = jsr THEN
databus <= wired_byte (pc (7 DOWNTO 0) );
adbus (7 DOWNTO 0) <= byte2;
write_mem <= '1'; WAIT FOR write_high_time;
write_mem <= '0'; WAIT FOR write_low_time;
databus <= "ZZZZZZZZ";
pc (7 DOWNTO 0) := inc (byte2);
ELSIF byte1 (7 DOWNTO 4) = bra THEN
IF ( byte1 (3) = '1' AND v = '1' ) OR ( byte1 (2) = '1' AND c = '1' ) OR
```

```
( byte1 (1) = '1' AND z = '1' ) OR ( byte1 (0) = '1' AND n = '1' )
THEN
pc (7 DOWNTO 0) := byte2;
END IF;
ELSE -- all other two-byte instructions
IF byte1 (4) = indirect THEN
adbus (11 DOWNTO 8) <= byte1 (3 DOWNTO 0);
adbus (7 DOWNTO 0) <= byte2;
read_mem <= '1'; WAIT FOR read_high_time;
byte2 := byte (databus);
read_mem <= '0'; WAIT FOR read_low_time;
END IF; -- ends indirect
IF byte1 (7 DOWNTO 5) = jmp THEN
pc := byte1 (3 DOWNTO 0) & byte2;
ELSIF byte1 (7 DOWNTO 5) = sta THEN
adbus <= byte1 (3 DOWNTO 0) & byte2;
databus <= wired_byte (ac);
write_mem <= '1'; WAIT FOR write_high_time;
write_mem <= '0'; WAIT FOR write_low_time;
databus <= "ZZZZZZZZ";
ELSE -- read operand for lda, and, add, sub
adbus (11 DOWNTO 8) <= byte1 (3 DOWNTO 0);
adbus (7 DOWNTO 0) <= byte2;
read_mem <= '1'; WAIT FOR read_high_time;
CASE byte1 (7 DOWNTO 5) IS
WHEN lda =>
ac := byte (databus);
WHEN ann =>
ac := ac AND byte (databus);
WHEN add =>
temp := add_cv (ac, byte (databus), c);
ac := temp (7 DOWNTO 0); c := temp (8); v := temp (9);
WHEN sbb =>
temp := sub_cv (ac, byte (databus), c);
ac := temp (7 DOWNTO 0); c := temp (8); v := temp (9);
WHEN OTHERS => NULL;
END CASE;
IF ac = zero_8 THEN z := '1'; END IF;
n := ac (7);
read_mem <= '0'; WAIT FOR read_low_time;
END IF; -- jmp / sta / lda, and, add, sub
END IF; -- jsr / bra / other double-byte instructions
END IF; -- single-byte / double-byte
END IF; -- interrupt / otherwise
END PROCESS;
END behavioral;
```