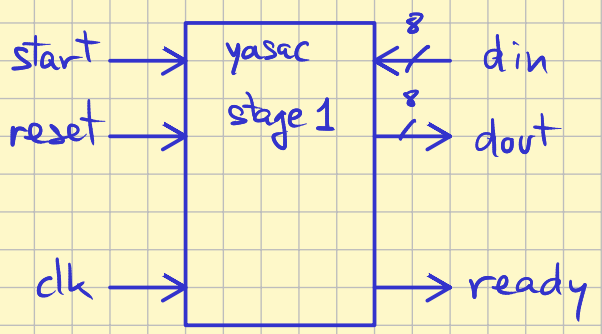
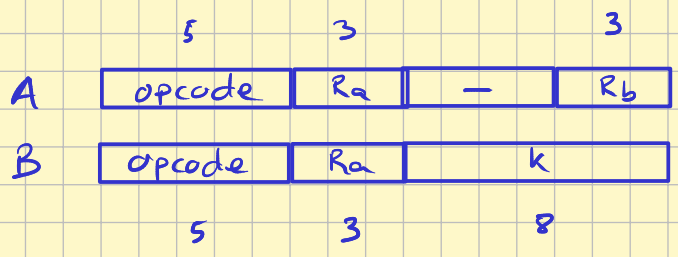
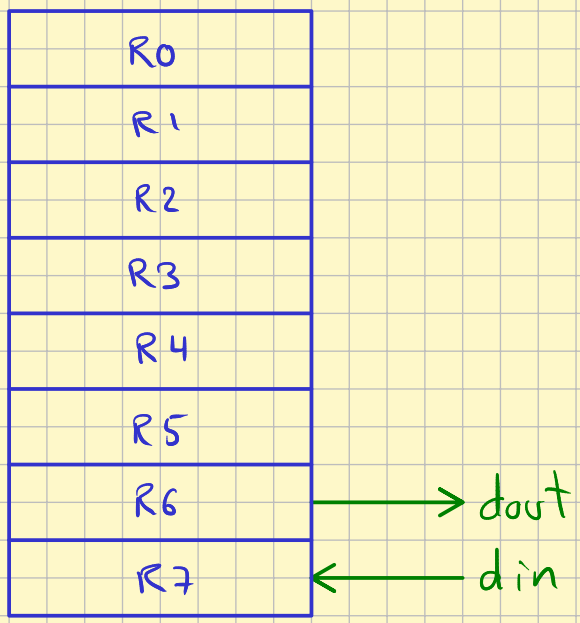


Instruction format



Registors

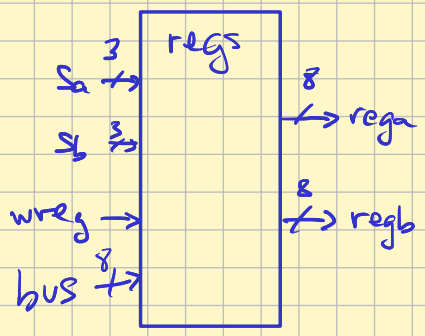


Instructions

```
LDI Ra, k      ; Ra ← k
MOV Ra, Rb     ; Ra ← Rb
ADD Ra, Rb     ; Ra ← Ra + Rb
SUB Ra, Rb     ; Ra ← Ra - Rb
STOP
```

Mnemonic	code
LDI	00001 (1)
MOV	00010 (2)
ADD	00011 (3)
SUB	00100 (4)
STOP	00101 (5)

Register array



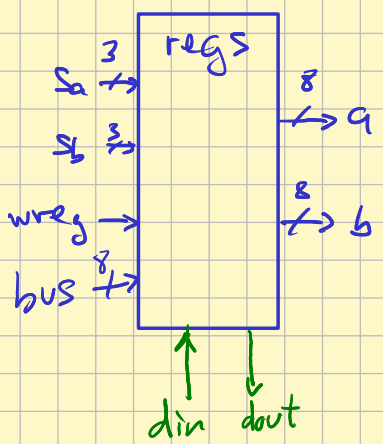
reg a = regs[sa]

regb = regs[sb]

wreg: regs[sa] ← bus

wreg	operation
0	INH
1	regs[sa] ← bus

* Basic (provisional) I/O system. (by modifying rega)
  - → Reading R_7 reads from din
  - → Writing R_6 writes dout



a = regs[sa]

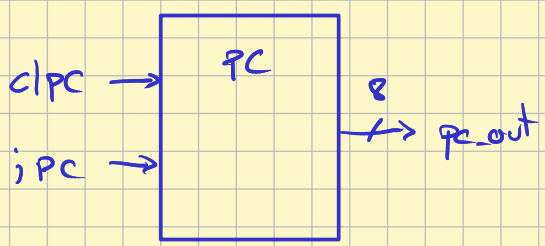
b = regs[sb]

wreg: regs[sa] ← bus

$\overline{wreg}$: regs[7] ← din

dout = regs[6]

Program counter (PC)

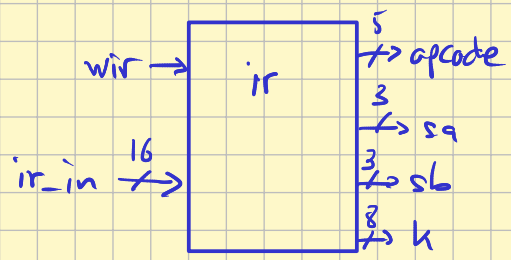


clpc: PC ← 0

incpc: PC ← PC + 1

clpc	incpc	PC ←
0	0	PC (INH.)
1	—	0
0	1	PC + 1

Instruction register (ir)



wir: ir ← ir_in

wir	ir ←
0	ir (INH)
1	ir_in

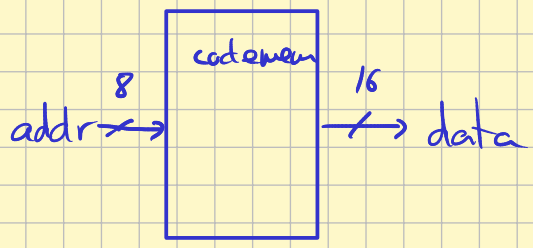
opcode = ir[15:11]

sa = ir[10:8]

sb = ir[2:0]

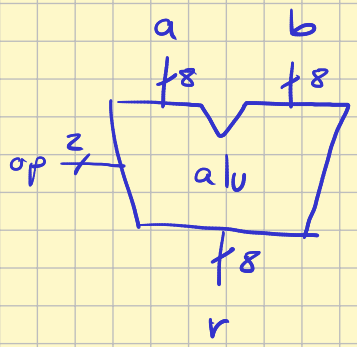
k = ir[7:0]

Code memory



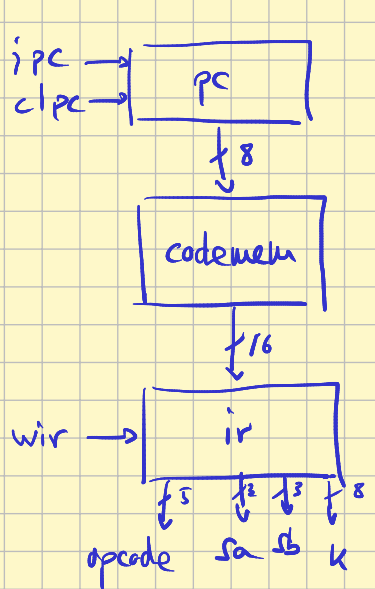
data = codemem[addr]

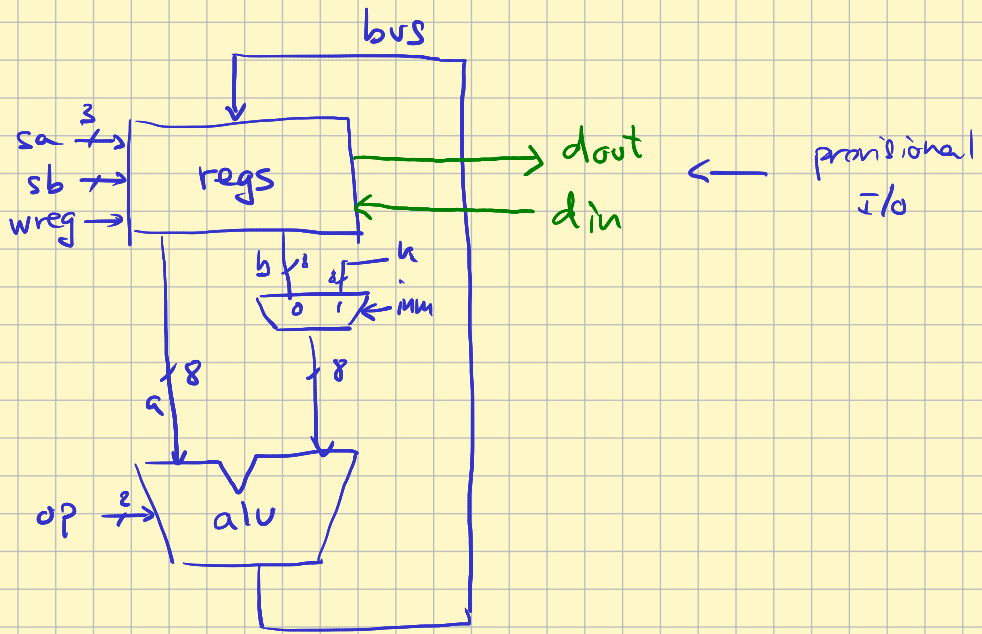
ALU



op	r
0 0	a+b
0 1	a
1 0	a-b
1 1	b

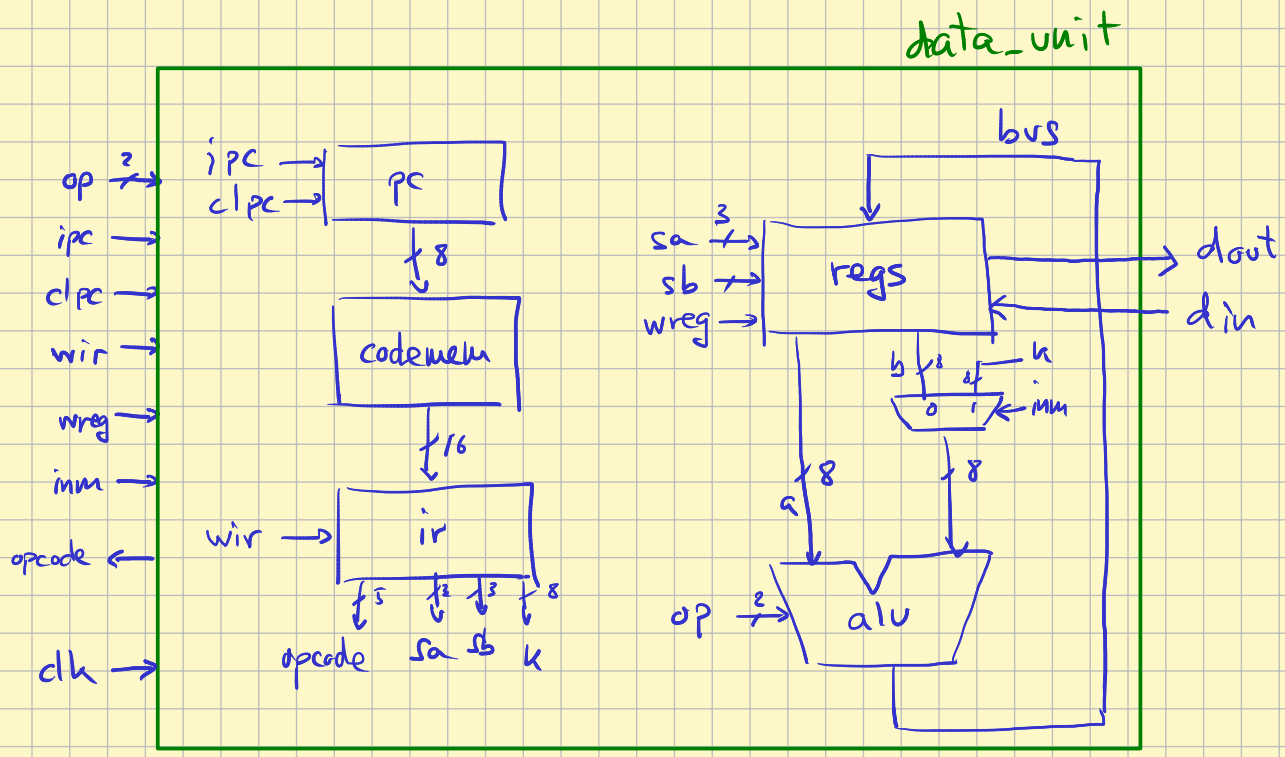
Data unit



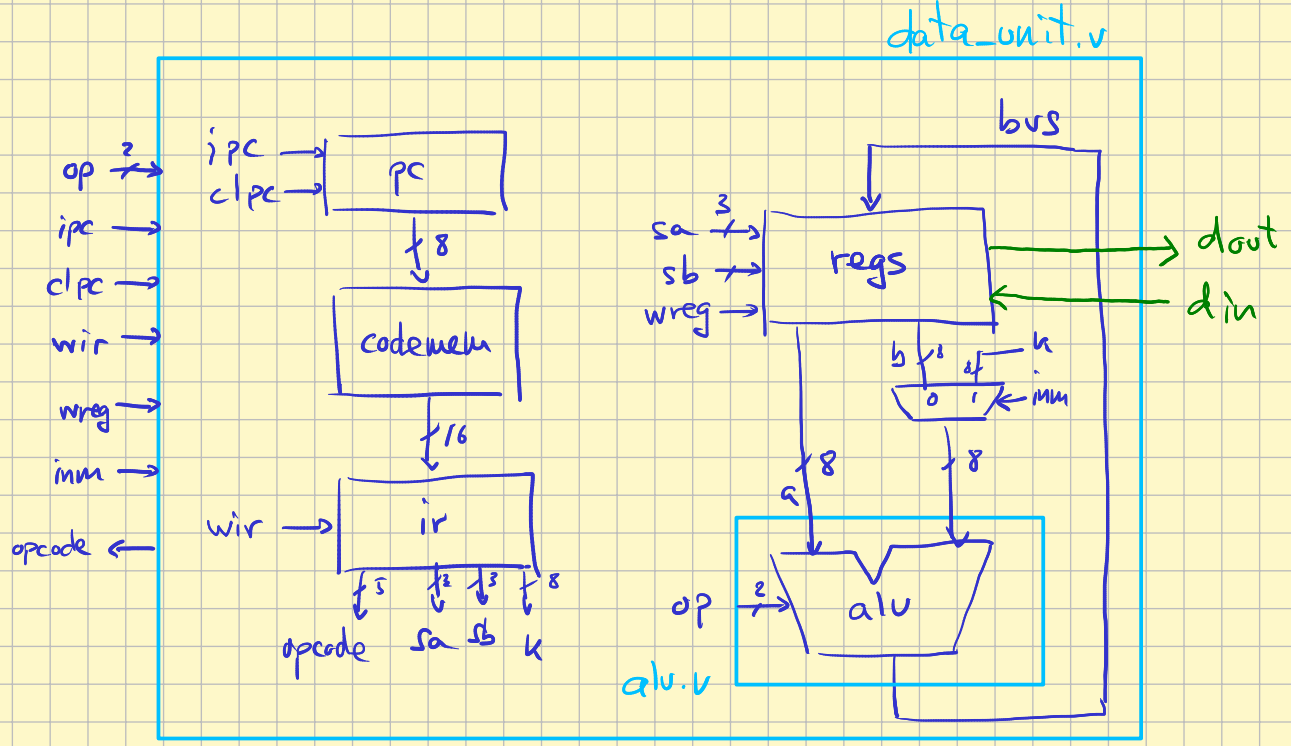


* Do you know how to design all the blocks?
* Can you describe each block in Verilog? (with a test bench?)

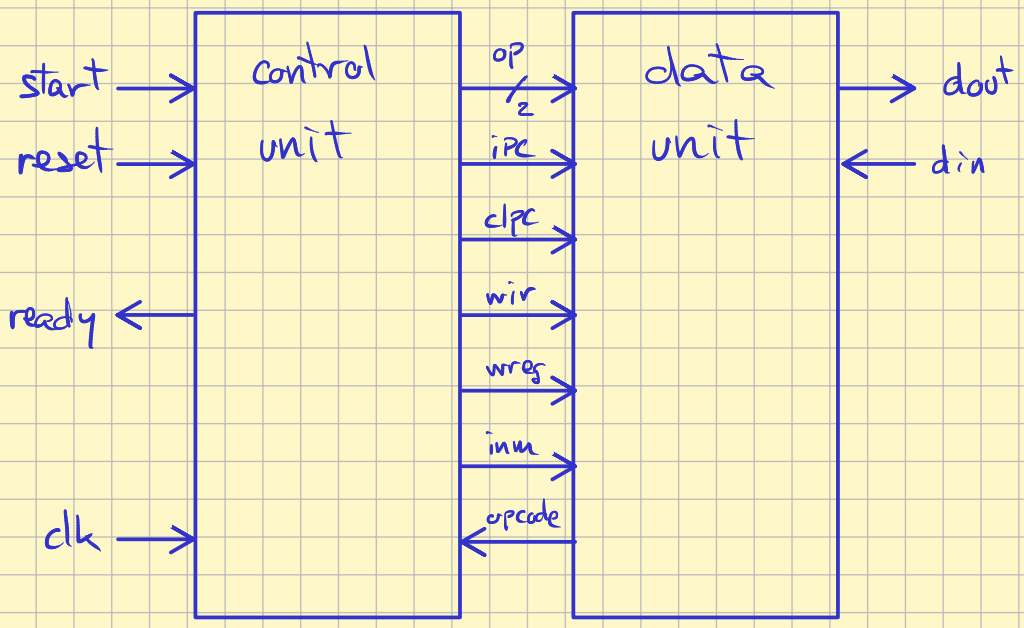
Data unit



Data unit



Control unit



Control unit.

External signals

- clk (in) clock
- reset (in) initialization
- start (in) start execution
- ready (out) ready indicator

Internal signals

- Control inputs to data unit.

Micro-operations

* Start: clear pc and continue to fetch pc ← 0 | cl_pc
* Fetch: read instruction to ir

  ir ← codemem[pc]; pc ← pc+1 | w_ir; i_pc

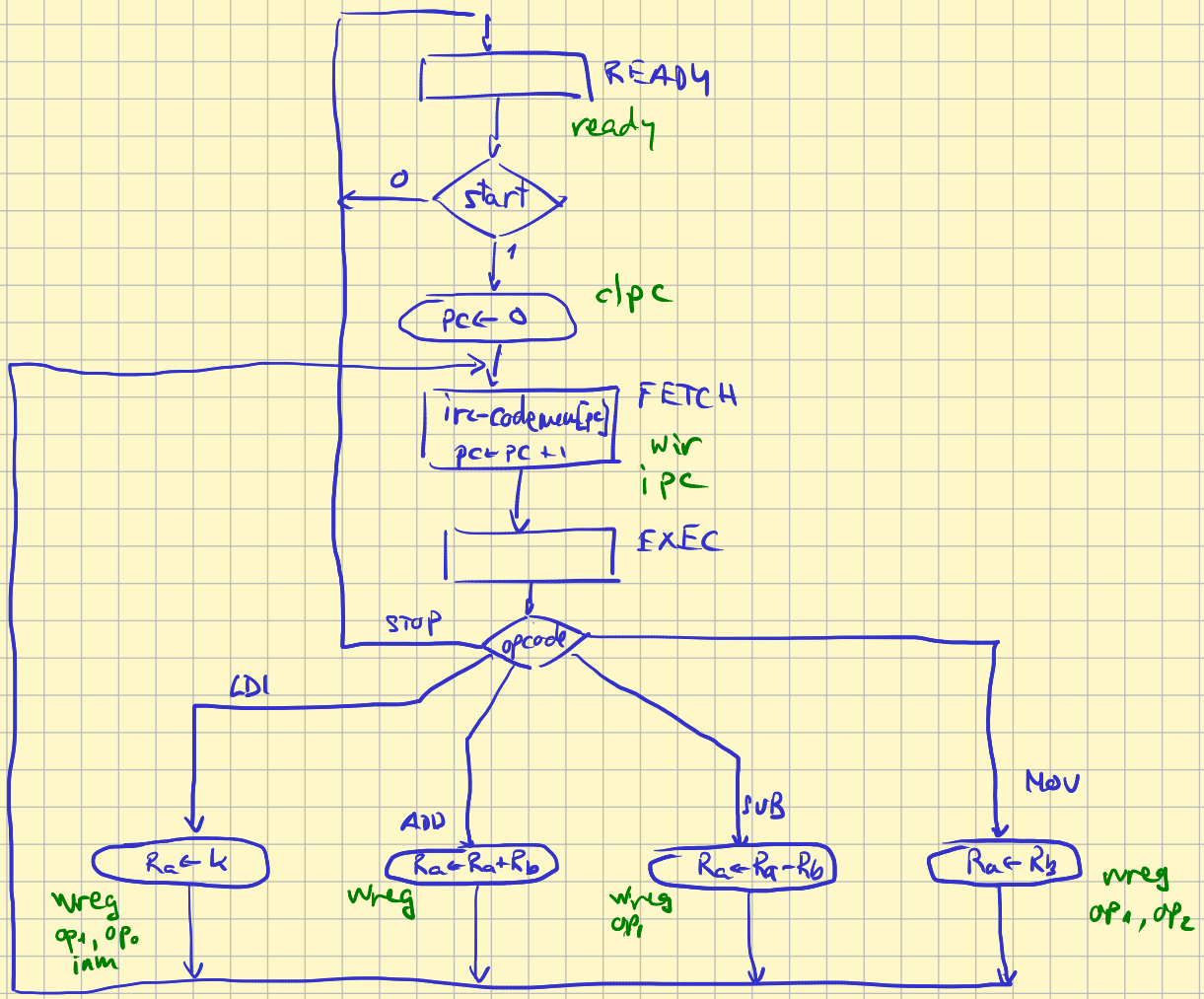
Execution depends on opcode (one state only)

ADD: Ra ← Ra+Rb | wreg; op=00
SUB: Ra ← Ra-Rb | wreg; op=10
MOV: Ra ← Rb | wreg; op=11
LDI: Ra ← k | wreg; inm; op=11

} Next state → FETCH

STOP: → next state: READY

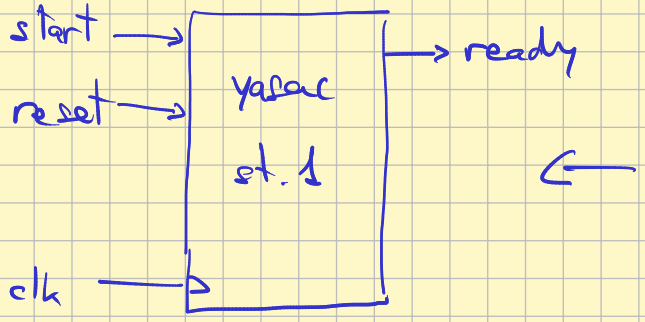
ASM chart

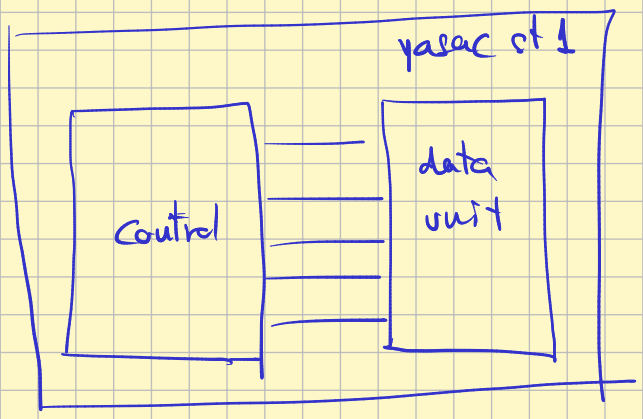


* Can you implement this FSM with a circuit?

* Can you describe this FSM in Verilog?

Testing





Test bench :

* Define a memory content (program)
* reset and start the computer.
* wait for "ready"
* check the result of the program.
* Not ok? check the waveforms.

Test program.

dout = 3 din + 5

```
MOV   R1, R7   // R1 = din
MOV   R0, R1   // R0 = din
ADD   R0, R1   // R0 = 2 din
LDI   R2, 5
SUB   R0, R2   // R0 = 2 din - 5
MOV   R6, R0   // dout = 2 din + 5
STOP
```

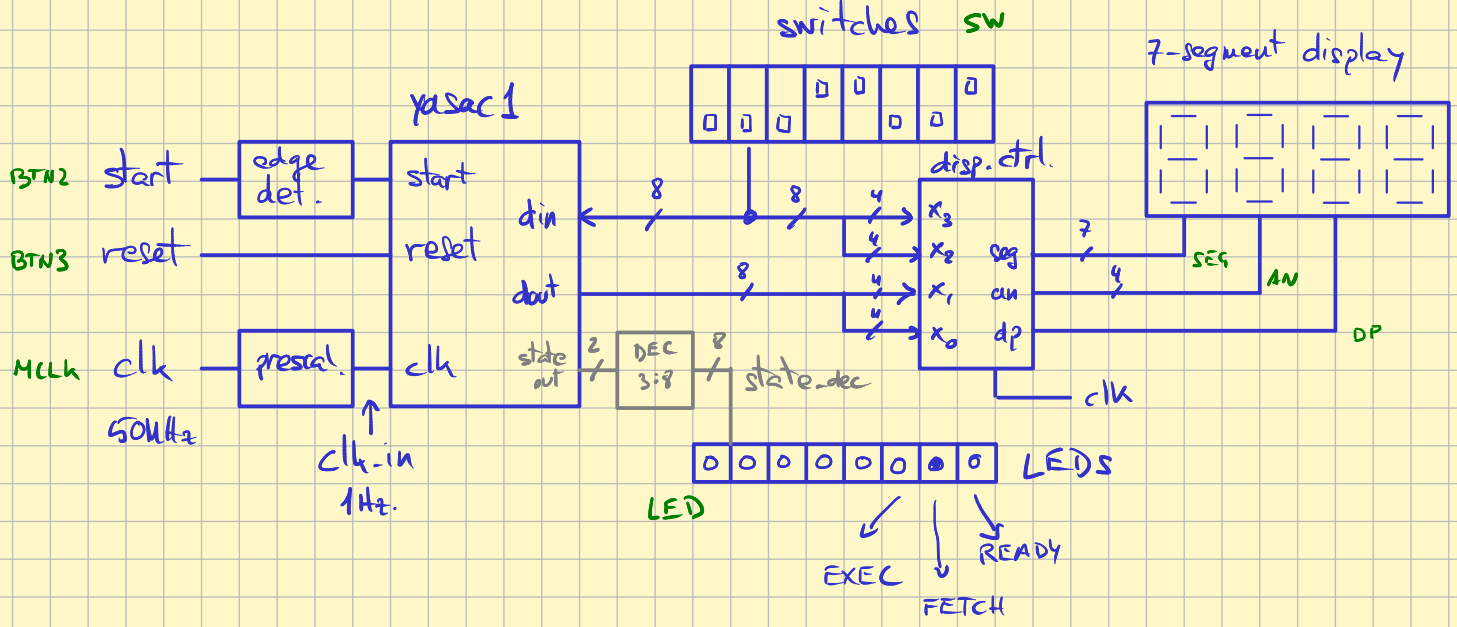
Test program 2

```
LDI   R1, 3
LDI   R2, 5
ADD   R1, R2
MOV   R6, R1
SUB   R6, R7
STOP           ; R6 = 8 - R7
```

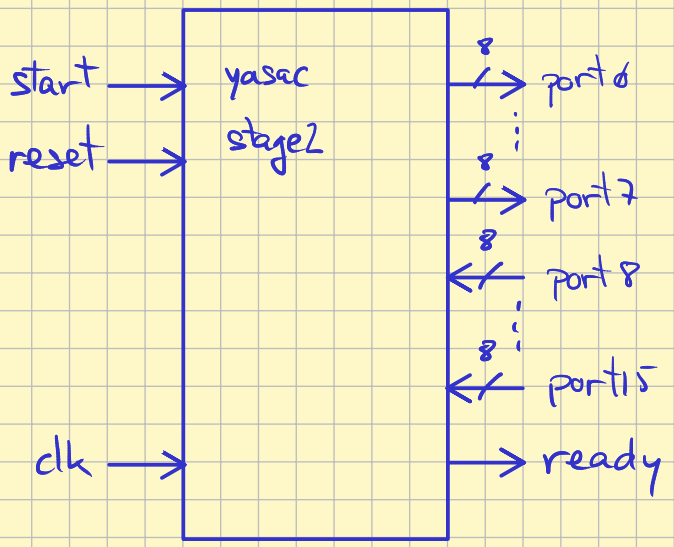
dout = 8 - din

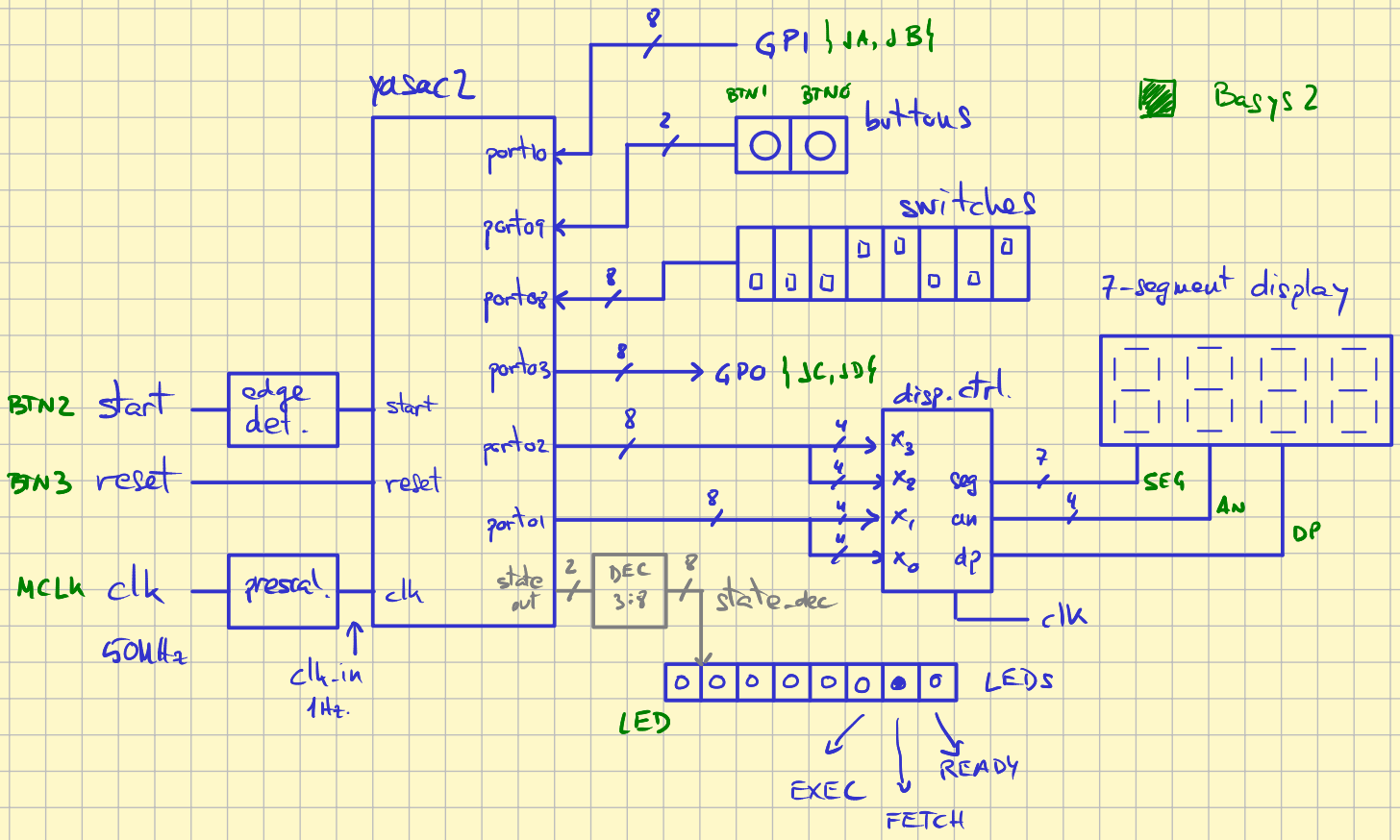
Implementation

system.v

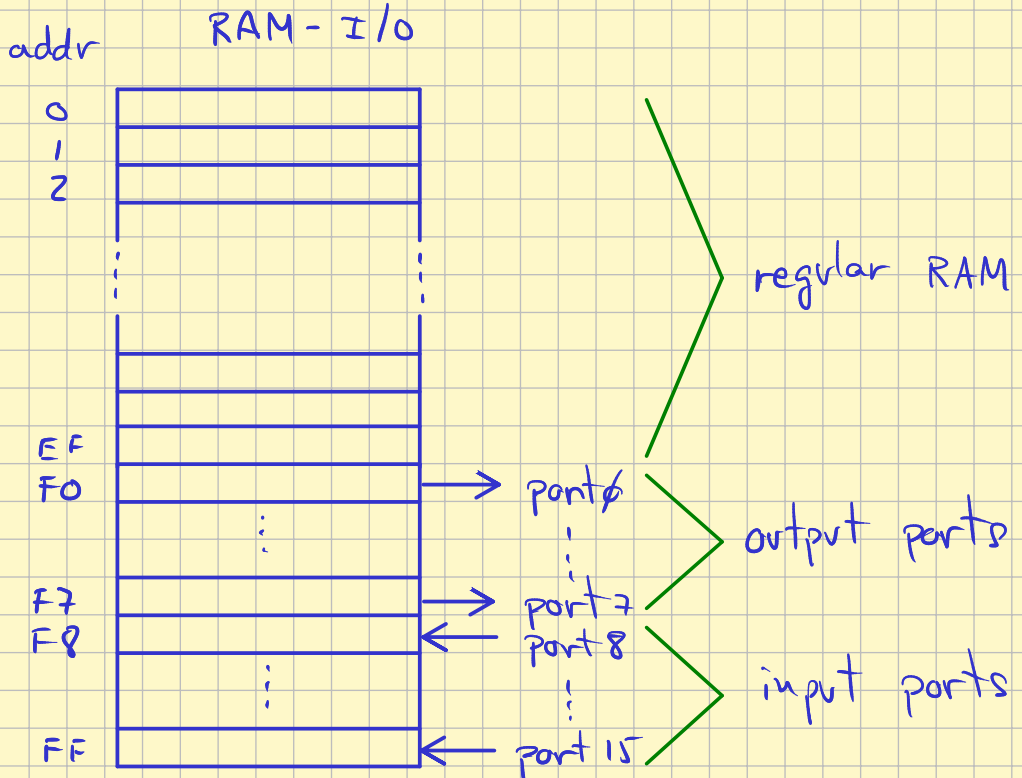


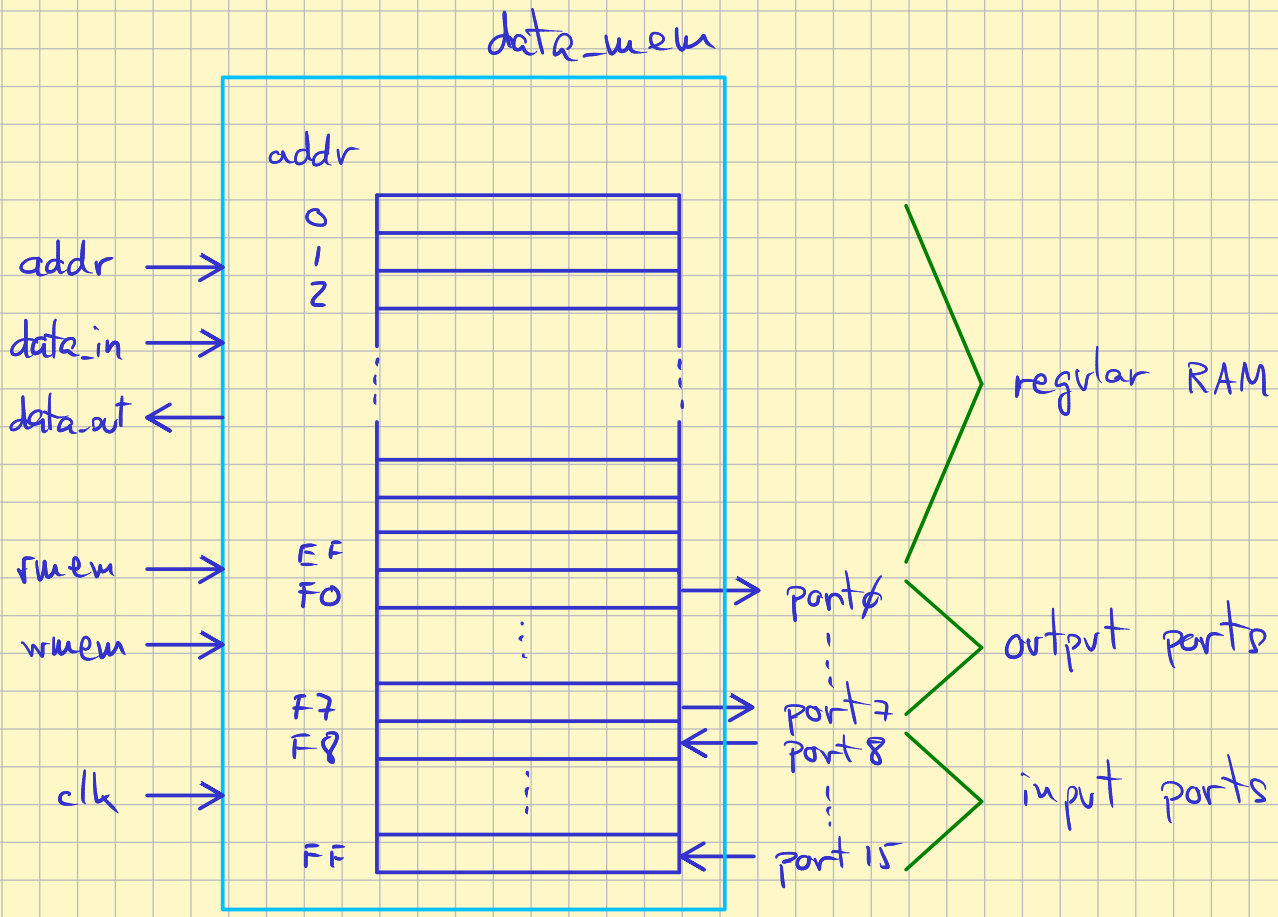
YASAC Stage 2



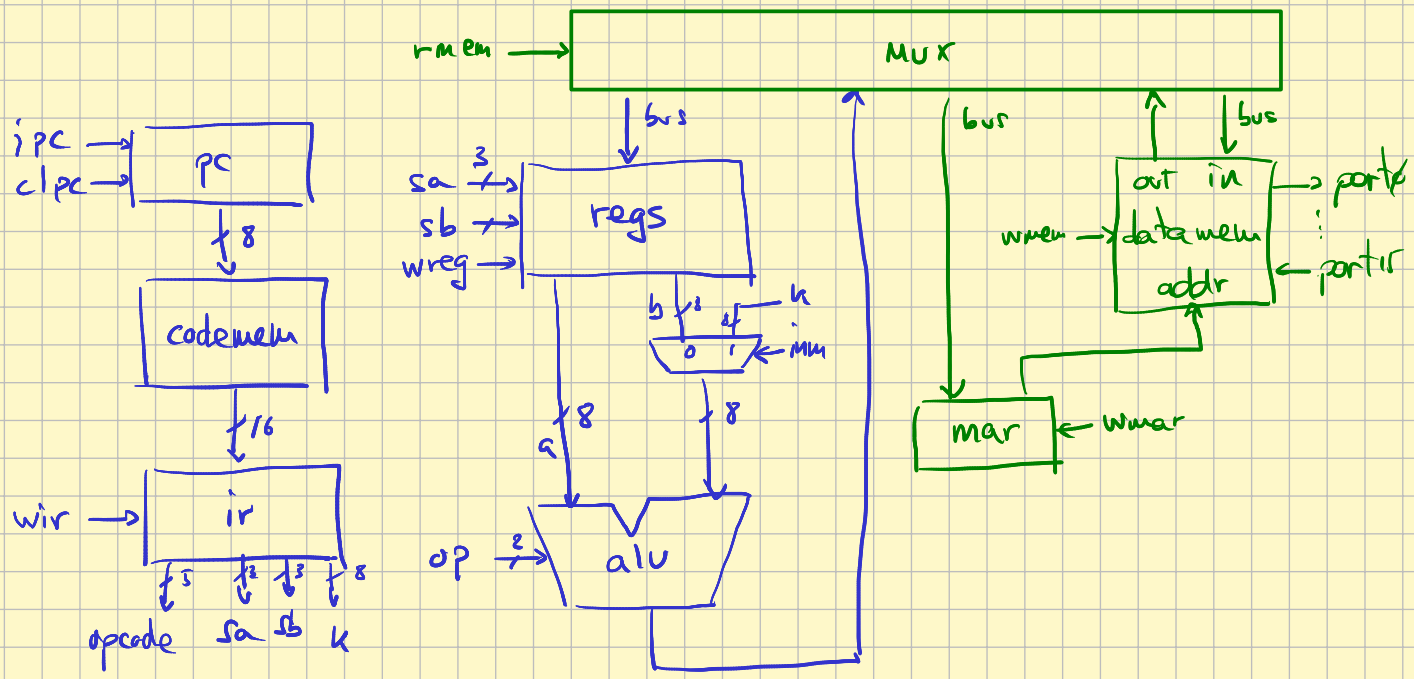


RAM - I/O module

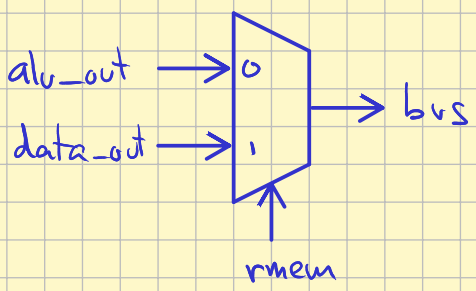




Stage 2 Data Unit



MUX →



Stage 2 macro-operations

LD Ra, Rb

1) mar ← rb — op=ALU_TRB, wmar
2) ra ← data_mem[mar] — rmem, wreg

ST Rb, Ra

1) mar ← rb — op=ALU_TRB, wmar
2) data_mem[mar] ← ra — op=ALU_TRA, wmem

LDS Ra, k

1) mar ← k — op=ALU_TRB, imm, wmar
2) ra ← data_mem[mar] — rmem, wreg

STS k, ra

1) mar ← k — op=ALU_TRB, imm, wmar
2) data_mem[mar] ← ra — op=ALU_TRA, wmem

States:

LDST1 : handles 1st μop of LD, ST, LDS, STS

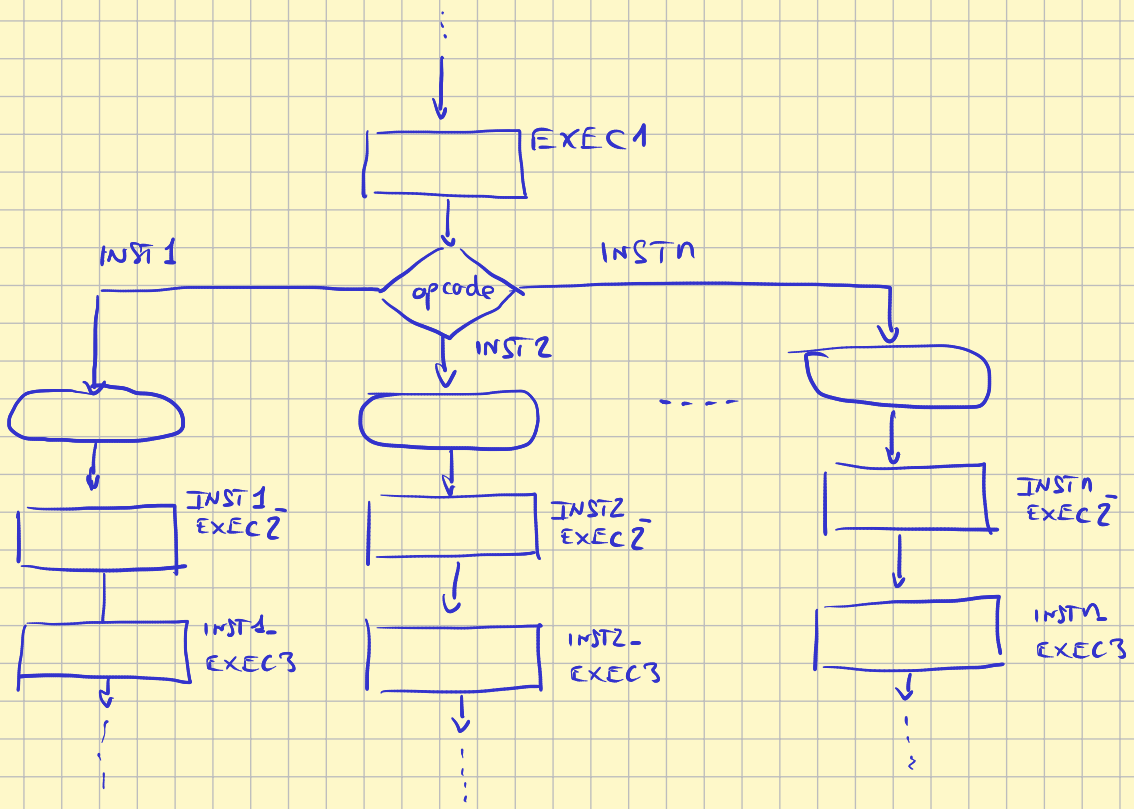
LDST2 : handles 2nd μop of LD, ST, LDS, STS

LDST1 : op=ALU_TRB, wmar
|||
EXEC LDS, STS : imm

LDST2 : LD, LDS : rmem, wreg
ST, STS : op=ALU_TRA, wmem

Control unit design approaches

Approach 1 Separate states for each instruction

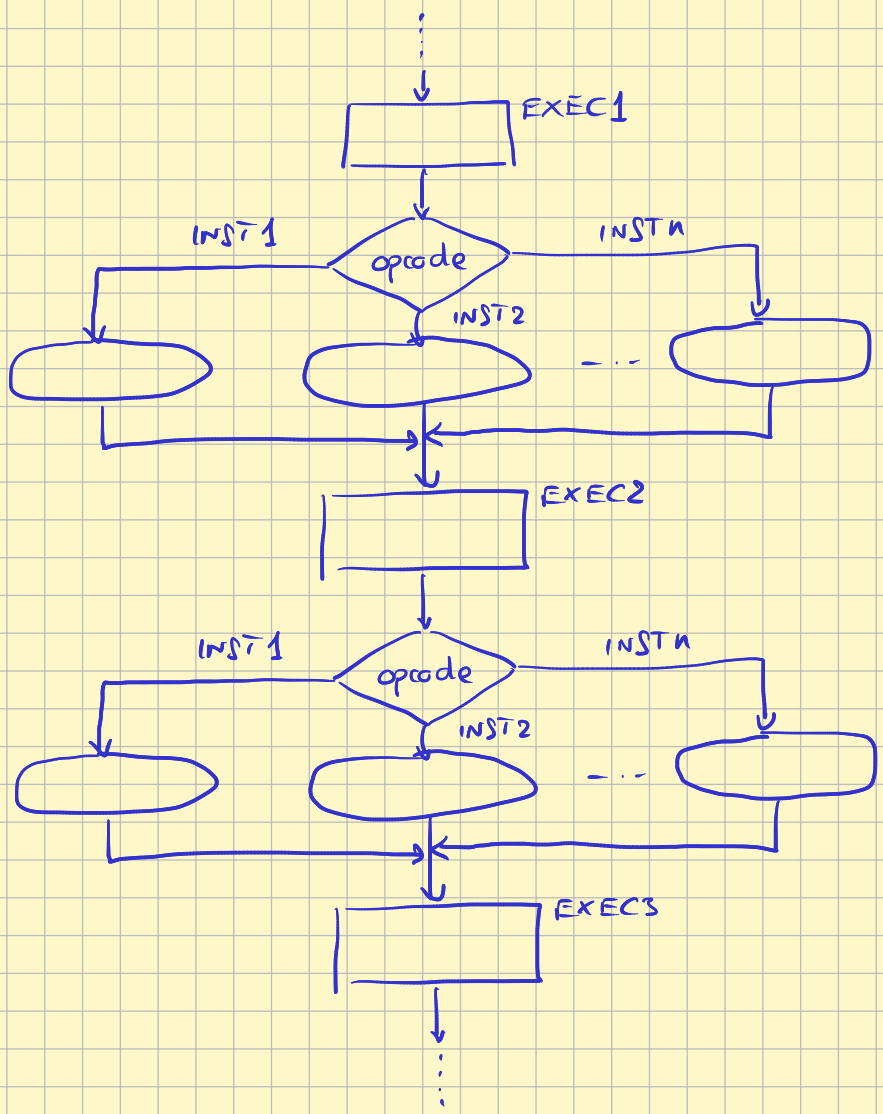


Good :
- new instructions do not modify (only a little) previous design (HDL code).
- easy to fix/modify individual instructions.
- easy to code in HDL.

Bad :
- Use more states than needed (similar instructions)
- Repeated code/hardware (similar instructions)

Approach 2 : Same states for all instruction

Do operation at each state depending on opcode.

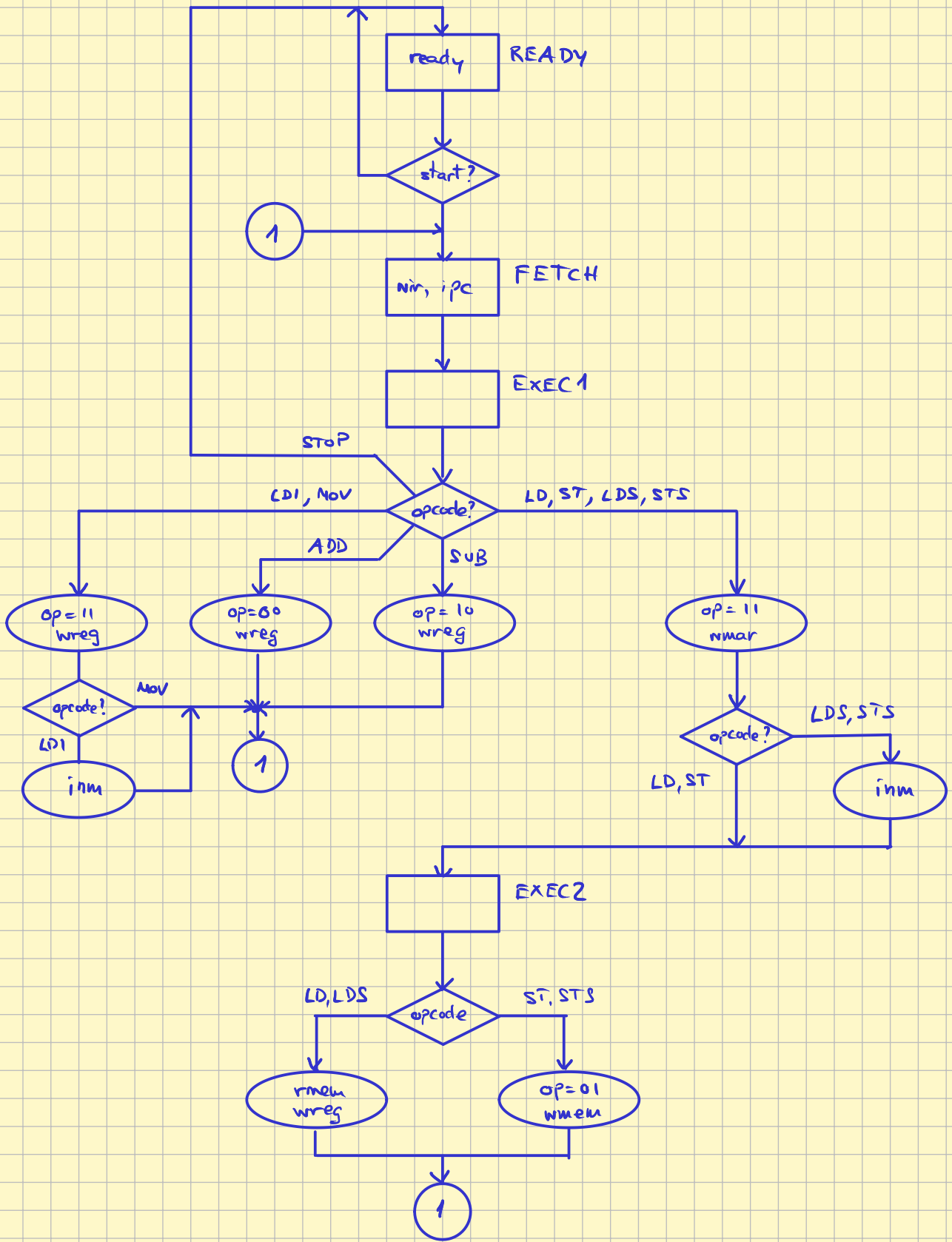


Good : - minimize states
- minimize logic (easier to optimize by logic syst. tools).
- easier to separate next state logic from control logic.

Bad : - HDL coding may not be as straightforward.
(better use and operation table)

Approach 3 : Mix both styles

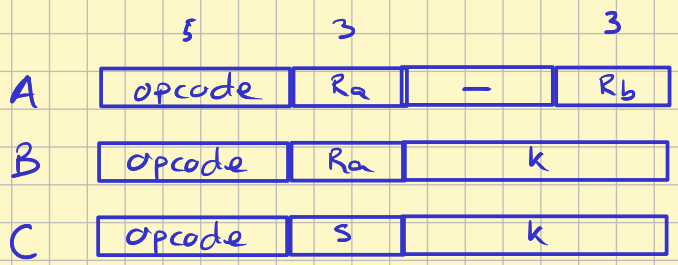
- Group similar instructions in the same "branch"
- Leave different branches for different groups of inst.


ready
READY
start?
1
nir, ipc
FETCH
EXEC1
STOP
opcode?
LDI, MOV
LD, ST, LDS, STS
ADD
SUB
op=11 wreg
op=00 wreg
op=10 wreg
op=11 wmar
opcode?
MOV
LDI
imm
1
opcode?
LDS, STS
LD, ST
imm
EXEC2
LD, LDS
opcode
ST, STS
rmem wreg
op=01 wmem
1

YASAC Stage 3

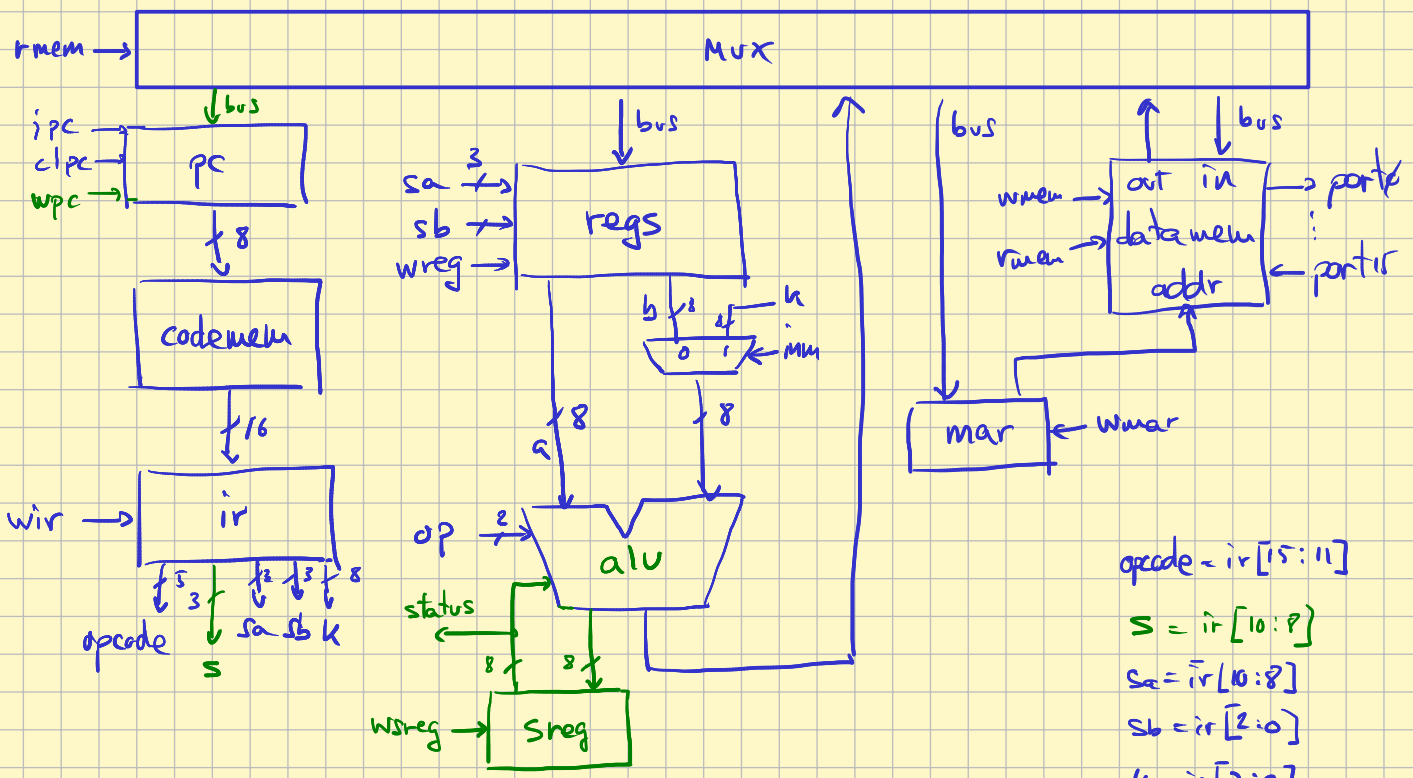
- Status register
- Branch instructions.

Instruction format



cond : - branch condition selector
- works as an extension of the opcode

Stage 3 data unit:



opcode = ir[15:11]

s = ir[10:8]

sa = ir[10:8]

sb = ir[2:0]

k = ir[7:0]

Changes:

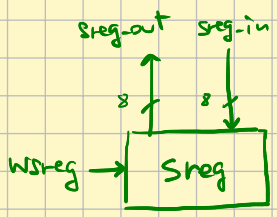
- The PC can be written (to allow jumping)
- The ALU generates S, V, N, Z, C state signals. (st_out)
- The ALU reads the status register in order to preserve some status bits.

Program counter

ipc: pc ← (pc + 1) mod 256

clpc: pc ← 0

wpc: pc ← bus



Status register

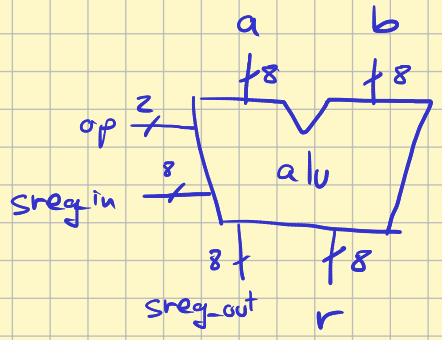
wsreg: sreg ← sreg_in

unconditional output.

Instruction register

s = ir[10:8]

ALU stage 3



op sym.	op	alu_out	sreg_out S V N Z C
ALU_ADD	0 0	a+b	* * * * *
ALU_TRA	0 1	a	- - - - -
ALU_SUB	1 0	a-b	* * * * *
ALU_TRB	1 1	b	- - - - -

* updated
\- not changed.

sreg_in = {-, -, -, S_i, V_i, N_i, Z_i, C_i}

sreg_out = {-, -, -, S_0, V_0, N_0, Z_0, C_0}

ADD

$C_0 = a_7 b_7 + b_7 \bar{r}_7 + a_7 \bar{r}_7$

$Z_0 = \overline{OR(r)}$

$N_0 = r(7)$

$V_0 = a_7 b_7 \bar{r}_7 + \bar{a}_7 \bar{b}_7 r_7$

$S_0 = V_0 \oplus N_0$

SUB

$C_0 = \bar{a}_7 b_7 + b_7 r_7 + \bar{a}_7 r_7$

$Z_0 = \overline{OR(r)}$

$N_0 = r(7)$

$V_0 = a_7 \bar{b}_7 \bar{r}_7 + \bar{a}_7 b_7 r_7$

$S_0 = V_0 \oplus N_0$

$$S_0 = \begin{cases} N_0 & \text{if } V_0 = 0, \\ \bar{N}_0 & \text{if } V_0 = 1 \end{cases}$$

sreg[0] = CF (Carry / Borrow): ADD: unsigned result greater than $2^8 - 1$
SUB: unsigned result is < 0.

sreg[1] = VF (oVerflow): Signed ADD/SUB overflow.

sreg[2] = ZF (Zero): the result is zero.

sreg[3] = NF (Negative): the signed result is negative

sreg[4] = SF (Signed test): The result is negative even if overflow exists.

Branch instructions

JMP k	1) pc ← k	imm, op=ALU_TRB, wpc	
BRBS s,k	1) status[s] : pc ← k	status[s] : (imm, op=ALU_TRB, wpc)	
BRBC s,k	1) $\overline{\text{status[s]}}$: pc ← k	$\overline{\text{status[s]}}$: (imm, op=ALU_TRB, wpc)	

Branch instructions aliases (Pseudo instructions)

BRBS CF,k	BRCS	Branch if Carry Set	
	BRLO	Branch if Lower	(unsigned)
BRBS ZF,k	BRZS	Branch if Zero Set	
	BREQ	Branch if EQual	
BRBS NF,k	BRMI	Branch if MInus	
BRBS VF,k	BRVS	Branch if oVerflow Set	
BRBS SF,k	BRLT	Branch if Less Than	(signed)
BRBC CF,k	BRCC	Branch if Carry Cleared	
	BRSH	Branch if Same or Higher	(unsigned)
BRBC ZF,k	BRZC	Branch if Zero Cleared	
	BRNE	Branch if Not Equal	
BRBC NF,k	BRPL	Branch if PLus	
BRBC VF,k	BRVC	Branch if oVerflow Cleared	
BRBC SF,k	BRGE	Branch if Greater or Equal	(signed)

Updated arithmetic instructions

ADD Ra,Rb	1) Ra ← Ra+Rb ; sreg ← sreg_out	op=ALU_ADD, wreg, wsreg
SUB Ra,Rb	1) Ra ← Ra-Rb ; sreg ← sreg_out	op=ALU_SUB, wreg, wsreg

Verilog design plan for stage 3

1) Update ALU: status output but no new operations.
   May write a t.b. for it.
2) Update data unit: PC, sreg, ALU connection.
3) Update control unit: JMP and BRBS/BRBC
4) Write a test program for JMP and BRBS / BRBC

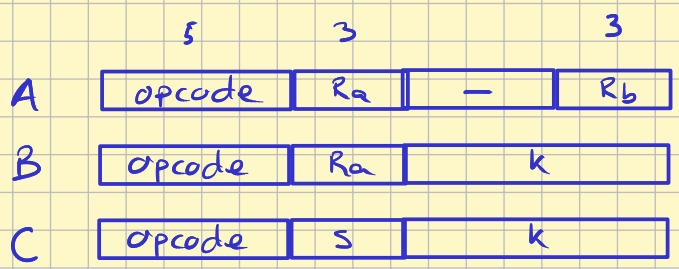
Test program for branch instructions

```
        LDI  R2, 0x50
        LDS  R1, 0xF8   // port08
        SUB  R1, R2
        BRES ZERO       // BRBS ZF, ZERO
        BRCS CARRY      // BRBS CF, CARRY
        BRVS OVER       // BRBS VF, OVER
        BRGE GREAT      // BRBC SF, GREAT
        JMP  END
ZERO:   LDI  R3, 1
        JMP  END
CARRY:  LDI  R3, 2
        JMP  END
OVER:   LDI  R3, 3
        JMP  END
GREAT:  LDI  R3, 4
END :   STS  0x81, R3   // port01
        STOP
```

YASAC Stage 4

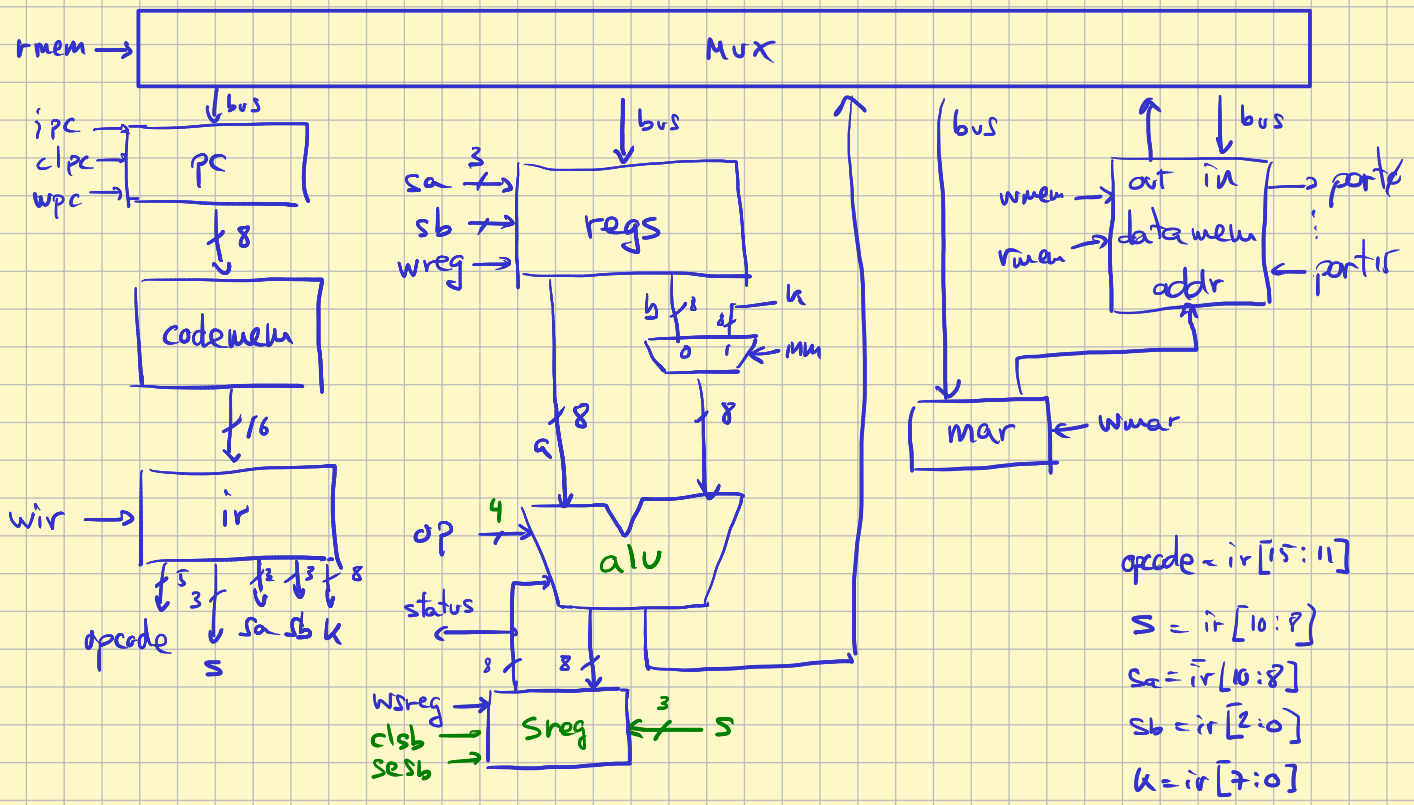
- Status register instructions
- Shift and logic instructions

Instruction format (same as in stage 3)



s : - branch condition selector
- BCLR, BSET flag selector.
- works as an extension of the opcode

Stage 4 data unit:



Changes:

* ALU has new operations (op is 4 bits now)
* Status bits can be modified through s, clsb, sesb.

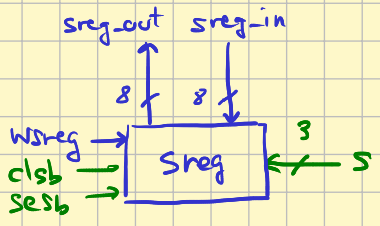
Status register

wsreg : sreg ← sreg_in

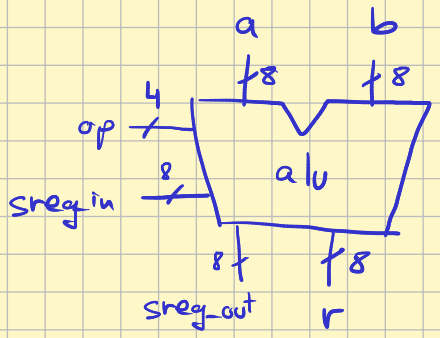
clsb : sreg[s] ← 0

sesb : sreg[s] ← 1

unconditional output.



ALU stage 4



OP sym.	op	r	sreg S V N Z C
ALU_ADD	0 0 0 0	a+b	* * * * *
ALU_TRA	0 0 0 1	a	- - - - -
ALU_SUB	0 0 1 0	a-b	* * * * *
ALU_TRB	0 0 1 1	b	- - - - -
ALU_NEG	0 1 0 0	-a	* * * * *
ALU_AND	0 1 0 1	AND(a,b)	* * * * -
ALU_OR	0 1 1 0	OR(a,b)	* * * * -
ALU_EOR	0 1 1 1	EOR(a,b)	* * * * -
ALU_ROR	1 0 0 0	SHR(a,c_{in})	* * * * *
ALU_ROL	1 0 0 1	SHL(a,c_{in})	* * * * *

* updated

\- not changed.

sreg_in = {-, -, -, S_i, V_i, N_i, Z_i, C_i}

sreg_out = {-, -, -, S_o, V_o, N_o, Z_o, C_o}

ADD

$C_0 = a_7 b_7 + b_7 \bar{r}_7 + a_7 \bar{r}_7$

$Z_0 = \overline{OR(r)}$

$N_0 = r(7)$

$V_0 = a_7 b_7 \bar{r}_7 + \bar{a}_7 \bar{b}_7 r_7$

$S_0 = V_0 \oplus N_0$

SUB

$C_0 = \bar{a}_7 b_7 + b_7 r_7 + \bar{a}_7 r_7$

$Z_0 = \overline{OR(r)}$

$N_0 = r(7)$

$V_0 = a_7 \bar{b}_7 \bar{r}_7 + \bar{a}_7 b_7 r_7$

$S_0 = V_0 \oplus N_0$

NEG

$C_0 = \overline{OR(a)}$

$Z_0 = \overline{OR(r)}$

$N_0 = r(7)$

$V_0 = a_7 r_7 + \bar{a}_7 \bar{r}_7$

$S_0 = V_0 \oplus N_0$

AND, OR, EOR

$C_0 = C_i$

$Z_0 = \overline{OR(r)}$

$N_0 = r(7)$

$V_0 = 0$

$S_0 = V_0 \oplus N_0$

RGR

$C_0 = a_0$

$Z = \overline{OR(r)}$

$N = r(7)$

$V_0 = C_{in} \oplus a_0$

$S_0 = V_0 \oplus N_0$

ROL

$C_0 = a_7$

$Z = \overline{OR(r)}$

$N = r(7)$

$V_0 = a_7 \oplus a_6$

$S_0 = V_0 \oplus N_0$

Status register instructions

BCLR s	1)	sreg[s] ← 0	clsb
BSET s	1)	sreg[s] ← 1	sesb

Logic instructions

AND Ra,Rb	1)	Ra ← Ra & Rb ;	sreg ← sreg_out (op=ALU_AND, wreg, wsreg)
OR Ra, Rb	1)	Ra ← Ra | Rb ;	sreg ← sreg_out (op=ALU_OR, wreg, wsreg)
EOR Ra, Rb	1)	Ra ← Ra ^ Rb ;	sreg ← sreg_out (op=ALU_EOR, wreg, wsreg)

Shift instructions

ROR Ra	1)	Ra ← SHR(Ra, c) ;	sreg ← sreg_out (op=ALU_ROR, wreg, wsreg)
ROL Ra	1)	Ra ← SHL(Ra, c) ;	sreg ← sreg_out (op=ALU_ROL, wreg, wsreg)

Verilog design plan

1) Update ALU with new instructions.
2) Update ALU connection in data unit.
3) Update control unit.
4) Write test programs for the new istructions

Logic instructions test program

input: porto8 = 0x5a (01011010)

expected output: porto1 = 0x0a
porto2 = 0xde

```
LDI  R1, 0x0f
LDS  R0, f8        // porto8
AND  R0, R1        // R0 = 00001010
STS  0xf1, R0      // porto1 = 0a
LDI  R2, 0xC3      // R2 = 11000011
EOR  R1, R2        // R1 = 11001100
LDS  R0, 0xf8      // porto8
OR   R0, R1        // R0 = 11011011
STS  0xf2, R0      // porto2 = de
STOP
```

PASS

shift and sreg test program

input: porta8 = 0x5a = 01011010

expected output: portb1 = 2
portb2 = a

```
LDI  R0, 0x00        // clear all flags
SUB  R0, R0
LDS  R0, 0x48
ROL  R0
ROL  R0
ROL  R0
ROL  R0              // R0 = 10100010 = a2
STS  0x41, R0        // portb1 = a2
CLC
RUR  R0
CLC
RUR  R0
CLC
ROR  R0
SEC
ROR  R0              // R0 = 8a
STS  0x42, R0        // portb2 = 8a
STOP
```

PASS

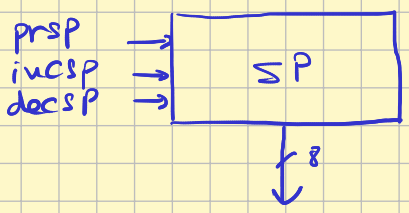
VASAC Stage 5

Stack and subroutines

New instructions CALL, RET, PUSH, PULL

Need:

* Stack pointer (sp) with increment, decrement and preset.
* PC reading to save its value.

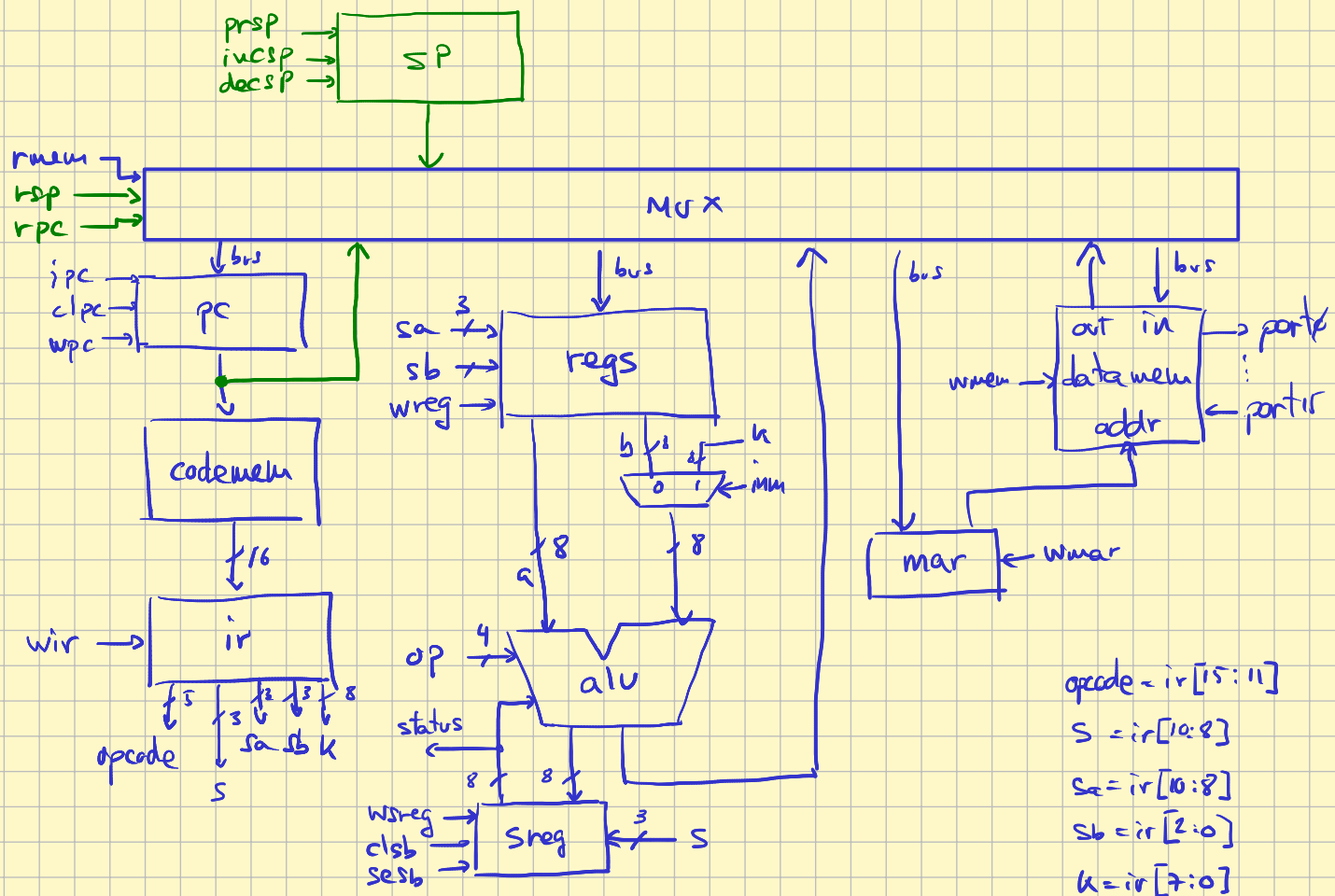


prsp : SP ← STACK_BASE

decsp : SP ← SP - 1

incsp : SP ← SP + 1

STACK_BASE = 0xEF



opcode = ir[15:11]

s = ir[10:8]

sa = ir[10:8]

sb = ir[2:0]

k = ir[7:0]

Instructions

PUSH Ra data_mem(sp) ← Ra ; sp ← sp - 1

1) mar ← sp ; sp ← sp - 1 rsp, wmar, decsp
2) data_mem(mar) ← Ra op = ALU_TRA, wmem

* sp ← sp-1 can be done in µop 2

POP Ra Ra ← data_mem(sp+1) ; sp ← sp + 1

1) sp ← sp + 1 incsp
2) mar ← sp rsp, wmar
3) Ra = data_mem(mar) rmem, wreg

CALL k data_mem(sp) ← pc ; pc ← k ; sp ← sp - 1

1) mar ← sp ; sp ← sp - 1 rsp, wmar, decsp
2) data_mem(mar) ← pc rpc, wmem
3) pc ← k inm, op = ALU_TRB, wpc

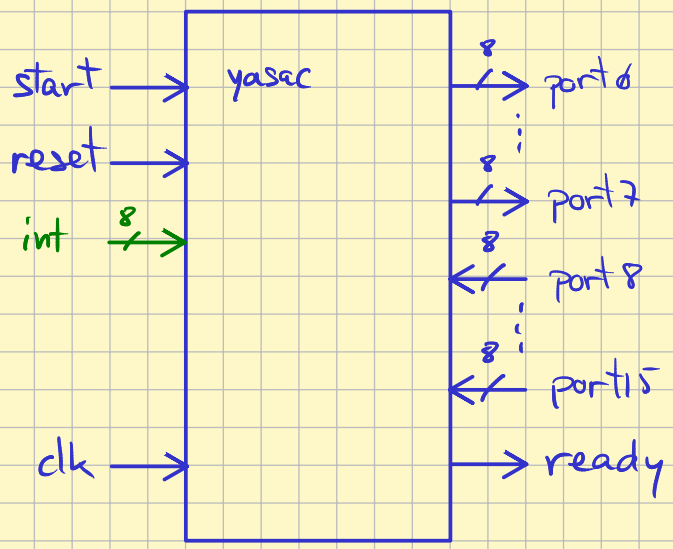
RET pc ← data_mem(sp+1) ; sp ← sp + 1

1) sp ← sp + 1 incsp
2) mar ← sp rsp, wmar
3) pc ← data_mem(mar) rmem, wpc

EXEC 1 same for: (PUSH, CALL) (POP, RET)

EXEC 2 same for: (POP, RET)

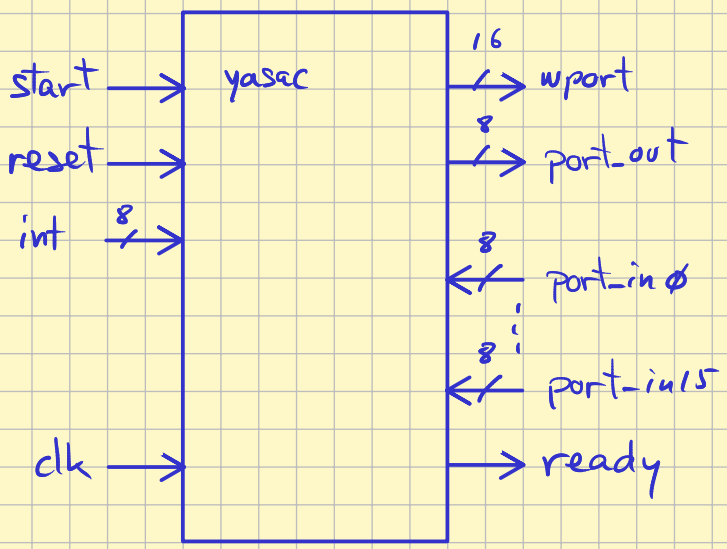
YASAC Stage 6. Interruptions.



vector	address	source	Description
0	0	RESET	Software reset
1	1	EXT_INT	External interrupt.
2	2	PIN_CHG0	Pin change associated to input port
3	3	PIN_CHG1	" " " " " "
4	4	TIMER	Timer interrupt
5	5	SPI_STC	SPI Serial transfer complete
6	6	USART_Rx	USART Rx complete
7	7	USART_Tx	USART Tx complete

Software reset: similar to global reset. May substitute global reset in a future version. Does not initilizes the stack pointer. No way to initialize SP from software, yet.

Alternative i/o 1 (memory-mapped)



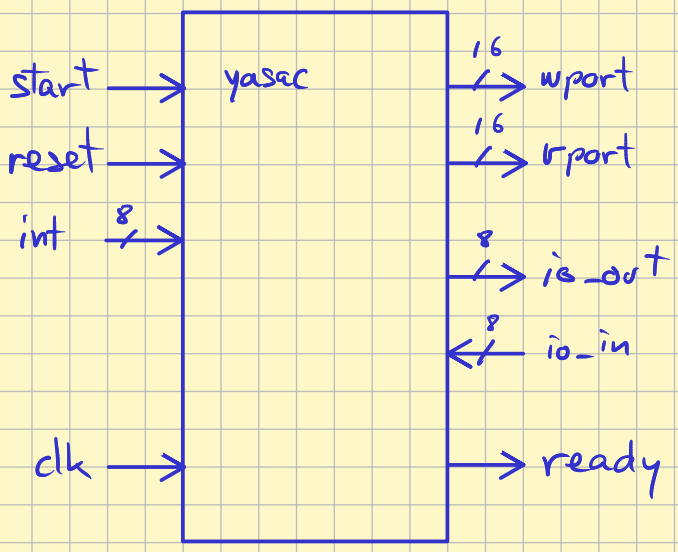
Good

* All ports are I/O
* Easy connection of peripherals: no need of intercomc. network
* Easy connection of internal registers (PC, SP, Sreg, ...)

Bad

* Not registered: need external registers working as i/o modules.

Alternative i/o 2 (memory-mapped)

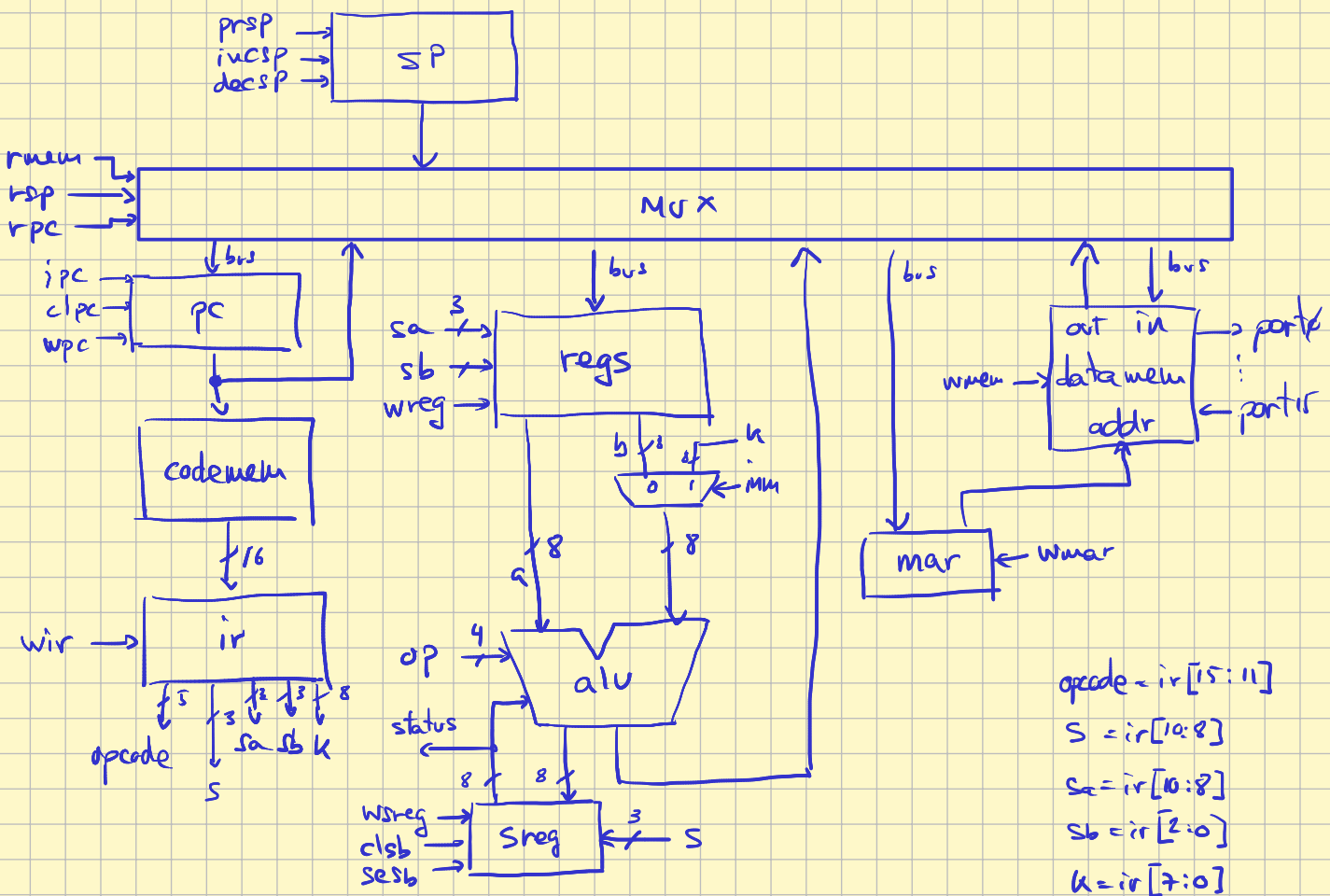


Good

* All ports are I/O
* Fewer i/o lines
* Easy connection of internal registers (PC, SP, Sreg, ...)

Bad

* Not registered: need external registers working as i/o modules.
* Needs external interconnection network (or tri-state).


prsp
incsp
decsp
SP
rmem
rsp
rpc
MUX
bus
ipc
clpc
wpc
pc
codemem
16
wir
ir
5
3
2
3
8
opcode
s
sa
sb
k
sa
3
sb
wreg
regs
b
4
k
0
1
mm
8
8
op
4
alu
status
8
8
wsreg
clsb
sesb
Sreg
3
s
bus
bus
bus
out
in
portw
wmem
datamem
addr
portr
mar
wmar
opcode = ir[15:11]
S = ir[10:8]
Sa = ir[10:8]
Sb = ir[2:0]
k = ir[7:0]