Softcore VHDL FPz8

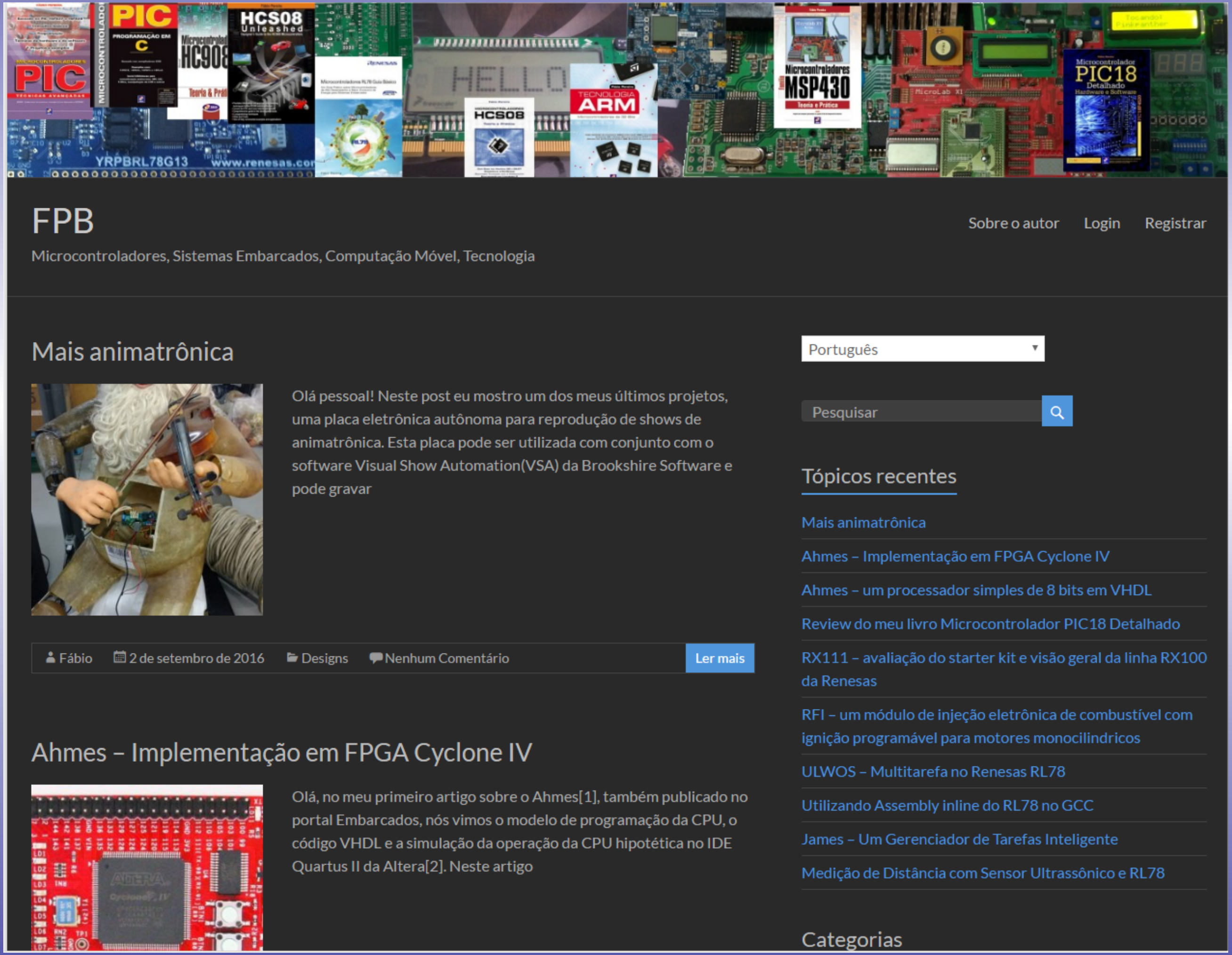
Desenvolvendo um microcontrolador em FPGA

Fábio Pereira 2017

Fábio Pereira

- Técnico em eletrônica, advogado, pós-graduado em projetos de dispositivos eletrônicos
- Autor de 9 livros na área de programação de microcontroladores (8, 16 e 32 bits), sendo 8 em português e 1 em inglês
- Fundador da ScTec, empresa que atuou por 10 anos na área de desenvolvimento em Joinville

Fábio Pereira



FPB

Microcontroladores, Sistemas Embarcados, Computação Móvel, Tecnologia

Sobre o autor Login Registrar

Mais animatrônica

Olá pessoal! Neste post eu mostro um dos meus últimos projetos, uma placa eletrônica autônoma para reprodução de shows de animatrônica. Esta placa pode ser utilizada com conjunto com o software Visual Show Automation(VSA) da Brookshire Software e pode gravar

Fábio 2 de setembro de 2016 Designs Nenhum Comentário Ler mais

Ahmes – Implementação em FPGA Cyclone IV

Olá, no meu primeiro artigo sobre o Ahmes[1], também publicado no portal Embarcados, nós vimos o modelo de programação da CPU, o código VHDL e a simulação da operação da CPU hipotética no IDE Quartus II da Altera[2]. Neste artigo

Português

Pesquisar

Tópicos recentes

- Mais animatrônica
- Ahmes – Implementação em FPGA Cyclone IV
- Ahmes – um processador simples de 8 bits em VHDL
- Review do meu livro Microcontrolador PIC18 Detalhado
- RX111 – avaliação do starter kit e visão geral da linha RX100 da Renesas
- RFI – um módulo de injeção eletrônica de combustível com ignição programável para motores monocilíndricos
- ULWOS – Multitarefa no Renesas RL78
- Utilizando Assembly inline do RL78 no GCC
- James – Um Gerenciador de Tarefas Inteligente
- Medição de Distância com Sensor Ultrassônico e RL78

Categorias

www.sctec.com.br/blog

Objetivos Primários

- Aprofundar estudos em VHDL e FPGAs
- Estudar a implementação de instruções e operações em CPUs
- Diversão!

Objetivos do Projeto

- Demonstrar a implementação de um core comercial em um FPGA utilizando VHDL (e aprender a fazer isso do zero)
- Desenvolver uma versão softcore de um microcontrolador comercial com in-circuit debugger e com suporte a programação via IDE e compilador de alto nível (C)
- Filosofia SoC

Zilog Z8 Encore

- Microcontrolador eficiente, relativamente rápido e com um sistema de depuração serial facilmente sintetizável
- Arquitetura Harvard de 8 bits sem acumulador e com até 4096 registradores endereçáveis pela CPU (mix de SFRs e GPRs (RAM))

Zilog Z8 Encore

- IDE (ZDS-II) simples, gratuita, pequena (menor que 30Mb), com compilador ANSI C completo e simulador/depurador

Zilog eZ8 - Arquitetura

- Arquitetura Harvard com três espaços de endereçamento: Programa, Registradores e Dados (não implementado em diversos modelos)
- 16 registradores de CPU (R0 a R15) selecionáveis entre 4096 (através do registrador RP)
- Accumulatorless: cada um dos 4096 registradores pode atuar como fonte ou destino de operações

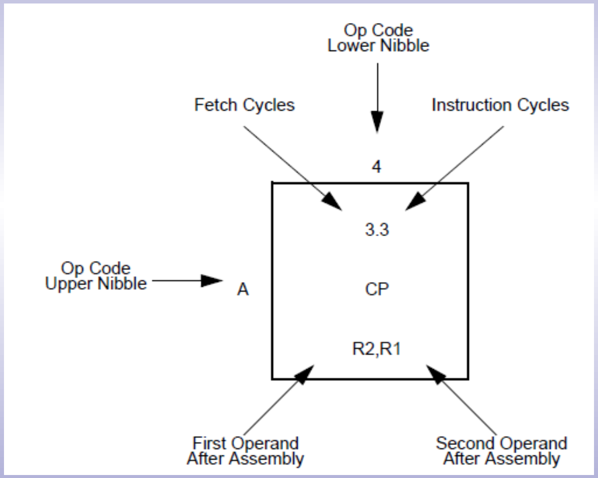
Zilog eZ8 - Arquitetura

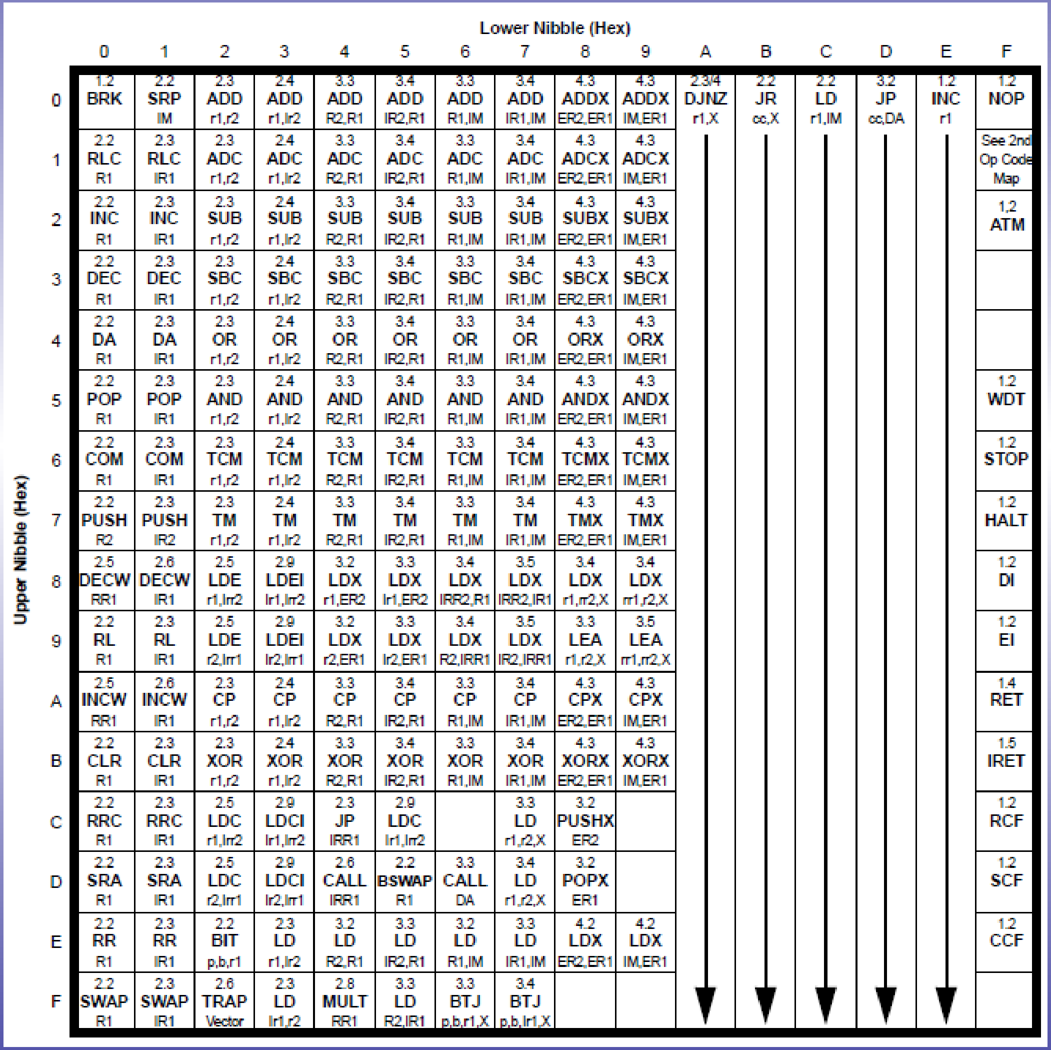
- 77 instruções (inclusive operações nativas de 16 bits)
- 6 modos de endereçamento:
  - Registrador (R) (4, 8 ou 12 bits)
  - Registrador Indireto (IR) (8 ou 12 bits)
  - Indexado (X)
  - Direto (DA)
  - Relativo (RA)
  - Imediato (IMM)

Zilog eZ8 - Arquitetura

- Modos de endereçamento envolvendo pares de registradores (para endereços de 12 ou 16 bits ou operandos de 16 bits)

Zilog eZ8 - Instruções



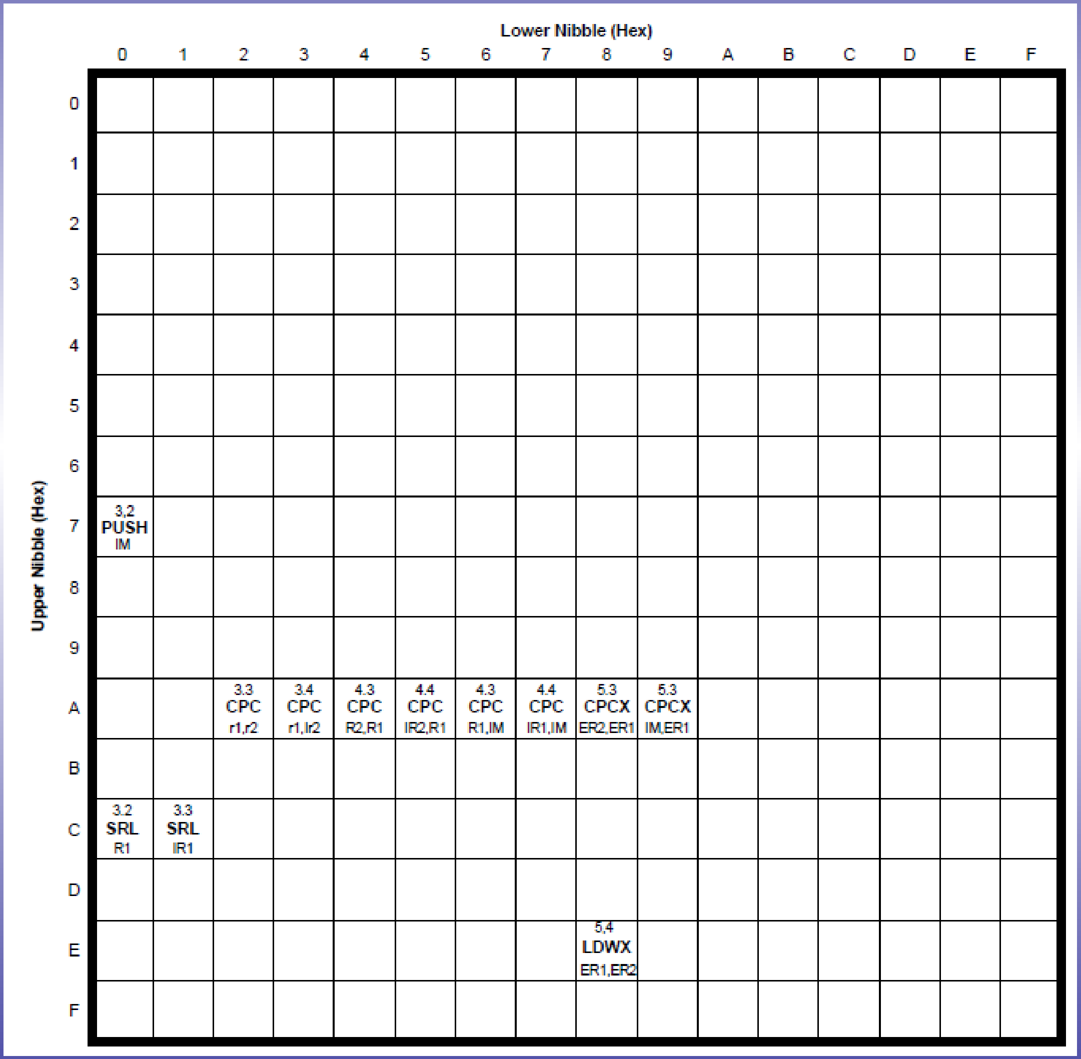


Lower Nibble (Hex)

Upper Nibble (Hex)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	1.2 BRK	2.2 SRP IM	2.3 ADD r1,r2	2.4 ADD r1,Ir2	3.3 ADD R2,R1	3.4 ADD IR2,R1	3.3 ADD R1,IM	3.4 ADD IR1,IM	4.3 ADDX ER2,ER1	4.3 ADDX IM,ER1	2.3/4 DJNZ r1,X	2.2 JR cc,X	2.2 LD r1,IM	3.2 JP cc,DA	1.2 INC r1	1.2 NOP
1	2.2 RLC R1	2.3 RLC IR1	2.3 ADC r1,r2	2.4 ADC r1,Ir2	3.3 ADC R2,R1	3.4 ADC IR2,R1	3.3 ADC R1,IM	3.4 ADC IR1,IM	4.3 ADCX ER2,ER1	4.3 ADCX IM,ER1						See 2nd Op Code Map
2	2.2 INC R1	2.3 INC IR1	2.3 SUB r1,r2	2.4 SUB r1,Ir2	3.3 SUB R2,R1	3.4 SUB IR2,R1	3.3 SUB R1,IM	3.4 SUB IR1,IM	4.3 SUBX ER2,ER1	4.3 SUBX IM,ER1						1,2 ATM
3	2.2 DEC R1	2.3 DEC IR1	2.3 SBC r1,r2	2.4 SBC r1,Ir2	3.3 SBC R2,R1	3.4 SBC IR2,R1	3.3 SBC R1,IM	3.4 SBC IR1,IM	4.3 SBCX ER2,ER1	4.3 SBCX IM,ER1						
4	2.2 DA R1	2.3 DA IR1	2.3 OR r1,r2	2.4 OR r1,Ir2	3.3 OR R2,R1	3.4 OR IR2,R1	3.3 OR R1,IM	3.4 OR IR1,IM	4.3 ORX ER2,ER1	4.3 ORX IM,ER1						
5	2.2 POP R1	2.3 POP IR1	2.3 AND r1,r2	2.4 AND r1,Ir2	3.3 AND R2,R1	3.4 AND IR2,R1	3.3 AND R1,IM	3.4 AND IR1,IM	4.3 ANDX ER2,ER1	4.3 ANDX IM,ER1						1.2 WDT
6	2.2 COM R1	2.3 COM IR1	2.3 TCM r1,r2	2.4 TCM r1,Ir2	3.3 TCM R2,R1	3.4 TCM IR2,R1	3.3 TCM R1,IM	3.4 TCM IR1,IM	4.3 TCMX ER2,ER1	4.3 TCMX IM,ER1						1.2 STOP
7	2.2 PUSH R2	2.3 PUSH IR2	2.3 TM r1,r2	2.4 TM r1,Ir2	3.3 TM R2,R1	3.4 TM IR2,R1	3.3 TM R1,IM	3.4 TM IR1,IM	4.3 TMX ER2,ER1	4.3 TMX IM,ER1						1.2 HALT
8	2.5 DECW RR1	2.6 DECW IR1	2.5 LDE r1,Irr2	2.9 LDEI Ir1,Irr2	3.2 LDX r1,ER2	3.3 LDX Ir1,ER2	3.4 LDX IRR2,R1	3.5 LDX IRR2,IR1	3.4 LDX r1,rr2,X	3.4 LDX rr1,r2,X						1.2 DI
9	2.2 RL R1	2.3 RL IR1	2.5 LDE r2,Irr1	2.9 LDEI Ir2,Irr1	3.2 LDX r2,ER1	3.3 LDX Ir2,ER1	3.4 LDX R2,IRR1	3.5 LDX IR2,IRR1	3.3 LEA r1,r2,X	3.5 LEA rr1,rr2,X						1.2 EI
A	2.5 INCW RR1	2.6 INCW IR1	2.3 CP r1,r2	2.4 CP r1,Ir2	3.3 CP R2,R1	3.4 CP IR2,R1	3.3 CP R1,IM	3.4 CP IR1,IM	4.3 CPX ER2,ER1	4.3 CPX IM,ER1						1.4 RET
B	2.2 CLR R1	2.3 CLR IR1	2.3 XOR r1,r2	2.4 XOR r1,Ir2	3.3 XOR R2,R1	3.4 XOR IR2,R1	3.3 XOR R1,IM	3.4 XOR IR1,IM	4.3 XORX ER2,ER1	4.3 XORX IM,ER1						1.5 IRET
C	2.2 RRC R1	2.3 RRC IR1	2.5 LDC r1,Irr2	2.9 LDCI Ir1,Irr2	2.3 JP IRR1	2.9 LDC Ir1,Irr2		3.3 LD r1,r2,X	3.2 PUSHX ER2							1.2 RCF
D	2.2 SRA R1	2.3 SRA IR1	2.5 LDC r2,Irr1	2.9 LDCI Ir2,Irr1	2.6 CALL IRR1	2.2 BSWAP R1	3.3 CALL DA	3.4 LD r1,r2,X	3.2 POPX ER1							1.2 SCF
E	2.2 RR R1	2.3 RR IR1	2.2 BIT p,b,r1	2.3 LD r1,Ir2	3.2 LD R2,R1	3.3 LD IR2,R1	3.2 LD R1,IM	3.3 LD IR1,IM	4.2 LDX ER2,ER1	4.2 LDX IM,ER1						1.2 CCF
F	2.2 SWAP R1	2.3 SWAP IR1	2.6 TRAP Vector	2.3 LD Ir1,r2	2.8 MULT RR1	3.3 LD R2,IR1	3.3 BTJ p,b,r1,X	3.4 BTJ p,b,Ir1,X								

Zilog eZ8 - Instruções



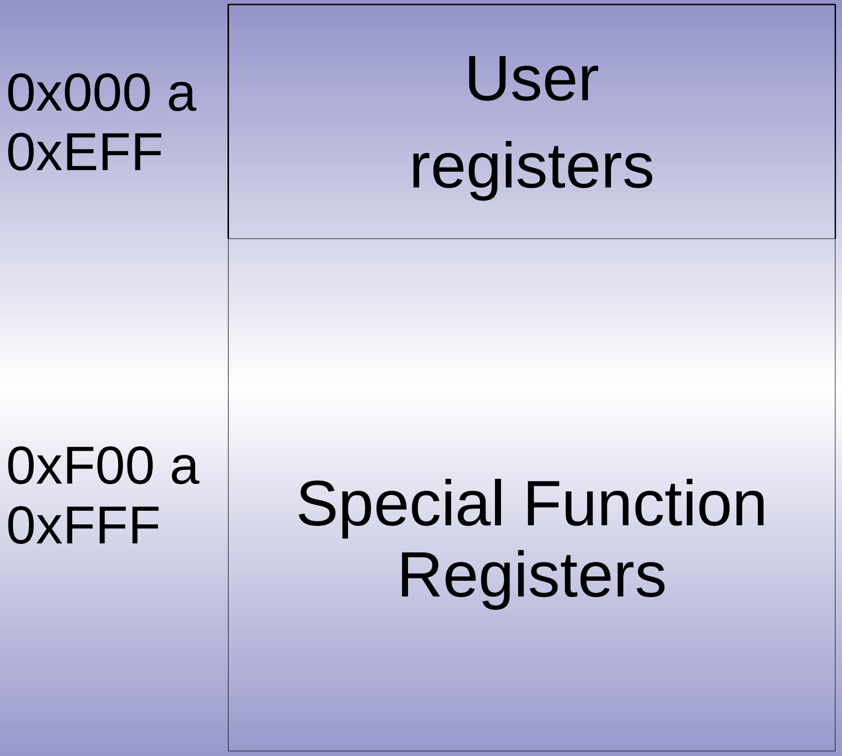
	Lower Nibble (Hex)															
Upper Nibble (Hex)	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0																
1																
2																
3																
4																
5																
6																
7	3,2 **PUSH** IM															
8																
9																
A			3.3 **CPC** r1,r2	3.4 **CPC** r1,Ir2	4.3 **CPC** R2,R1	4.4 **CPC** IR2,R1	4.3 **CPC** R1,IM	4.4 **CPC** IR1,IM	5.3 **CPCX** ER2,ER1	5.3 **CPCX** IM,ER1						
B																
C	3.2 **SRL** R1	3.3 **SRL** IR1														
D																
E									5,4 **LDWX** ER1,ER2							
F																

Zilog eZ8 – Mapa de Memória

0x0000	Option
0x0002	Reset vector
0x0004	WDT vector
0x0006	Illegal vector
0x0008 a 0x0037	Interrupt vectors
0x0038 a 0xFFFF	Program memory area

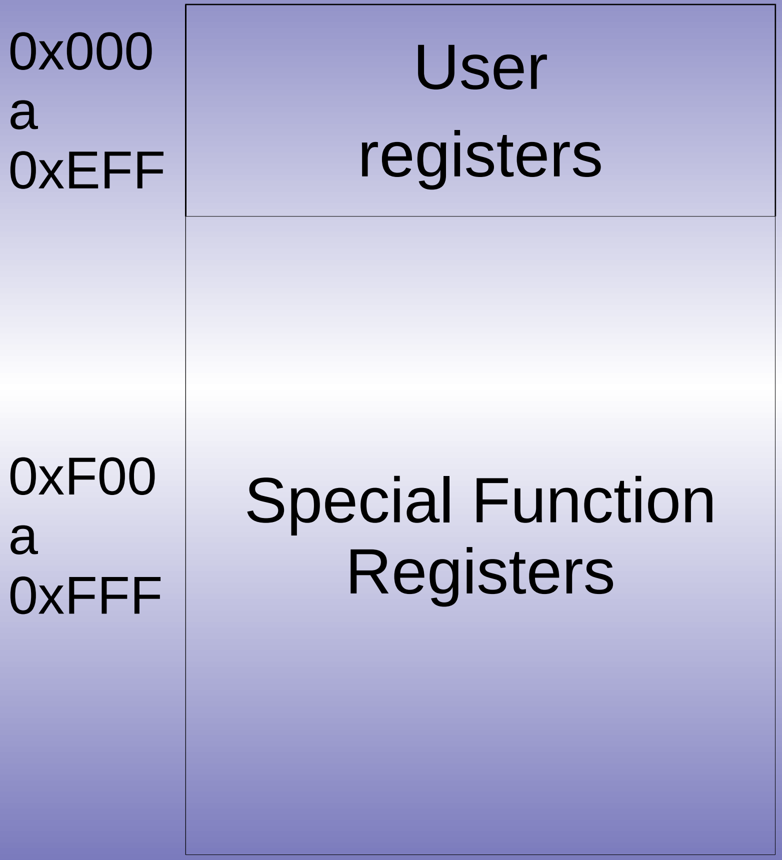
Memória de Programa

Zilog eZ8 – Mapa de Memória



Registradores

Zilog eZ8 – Mapa de Memória



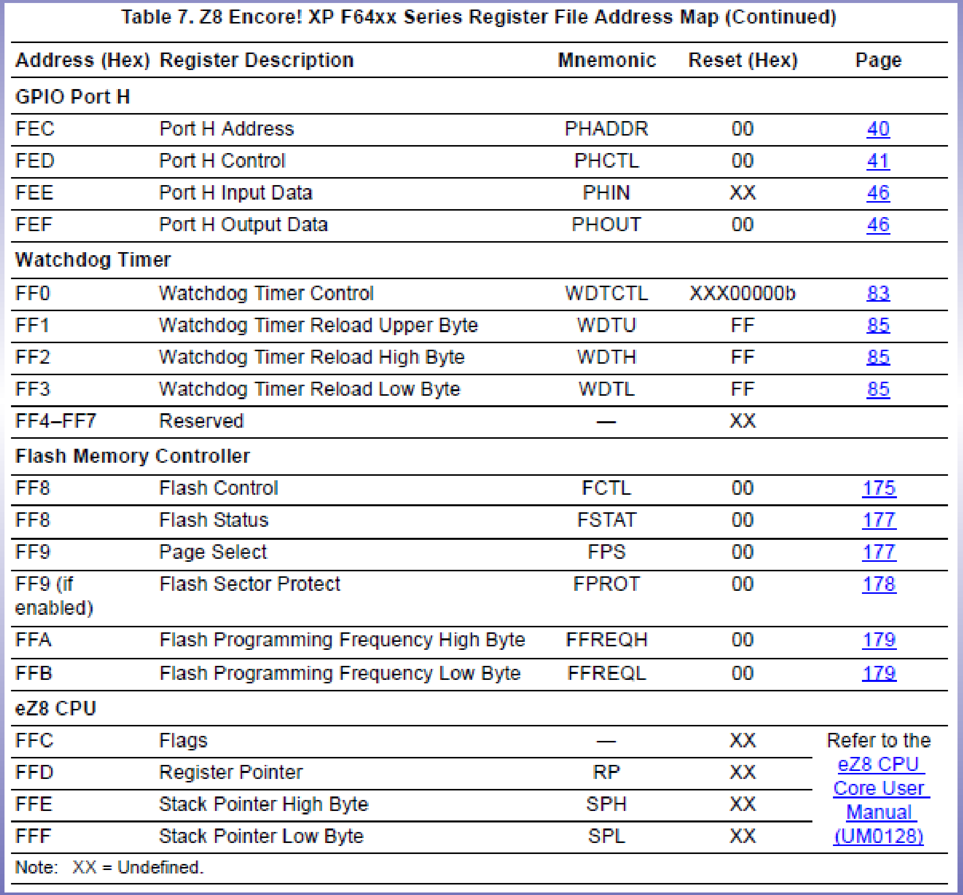


Table 7. Z8 Encore! XP F64xx Series Register File Address Map (Continued)

Address (Hex)	Register Description	Mnemonic	Reset (Hex)	Page
GPIO Port H				
FEC	Port H Address	PHADDR	00	40
FED	Port H Control	PHCTL	00	41
FEE	Port H Input Data	PHIN	XX	46
FEF	Port H Output Data	PHOUT	00	46
Watchdog Timer				
FF0	Watchdog Timer Control	WDTCTL	XXX00000b	83
FF1	Watchdog Timer Reload Upper Byte	WDTU	FF	85
FF2	Watchdog Timer Reload High Byte	WDTH	FF	85
FF3	Watchdog Timer Reload Low Byte	WDTL	FF	85
FF4–FF7	Reserved	—	XX	
Flash Memory Controller				
FF8	Flash Control	FCTL	00	175
FF8	Flash Status	FSTAT	00	177
FF9	Page Select	FPS	00	177
FF9 (if enabled)	Flash Sector Protect	FPROT	00	178
FFA	Flash Programming Frequency High Byte	FFREQH	00	179
FFB	Flash Programming Frequency Low Byte	FFREQL	00	179
eZ8 CPU				
FFC	Flags	—	XX	Refer to the eZ8 CPU Core User Manual (UM0128)
FFD	Register Pointer	RP	XX	
FFE	Stack Pointer High Byte	SPH	XX	
FFF	Stack Pointer Low Byte	SPL	XX	

Note: XX = Undefined.

Registradores

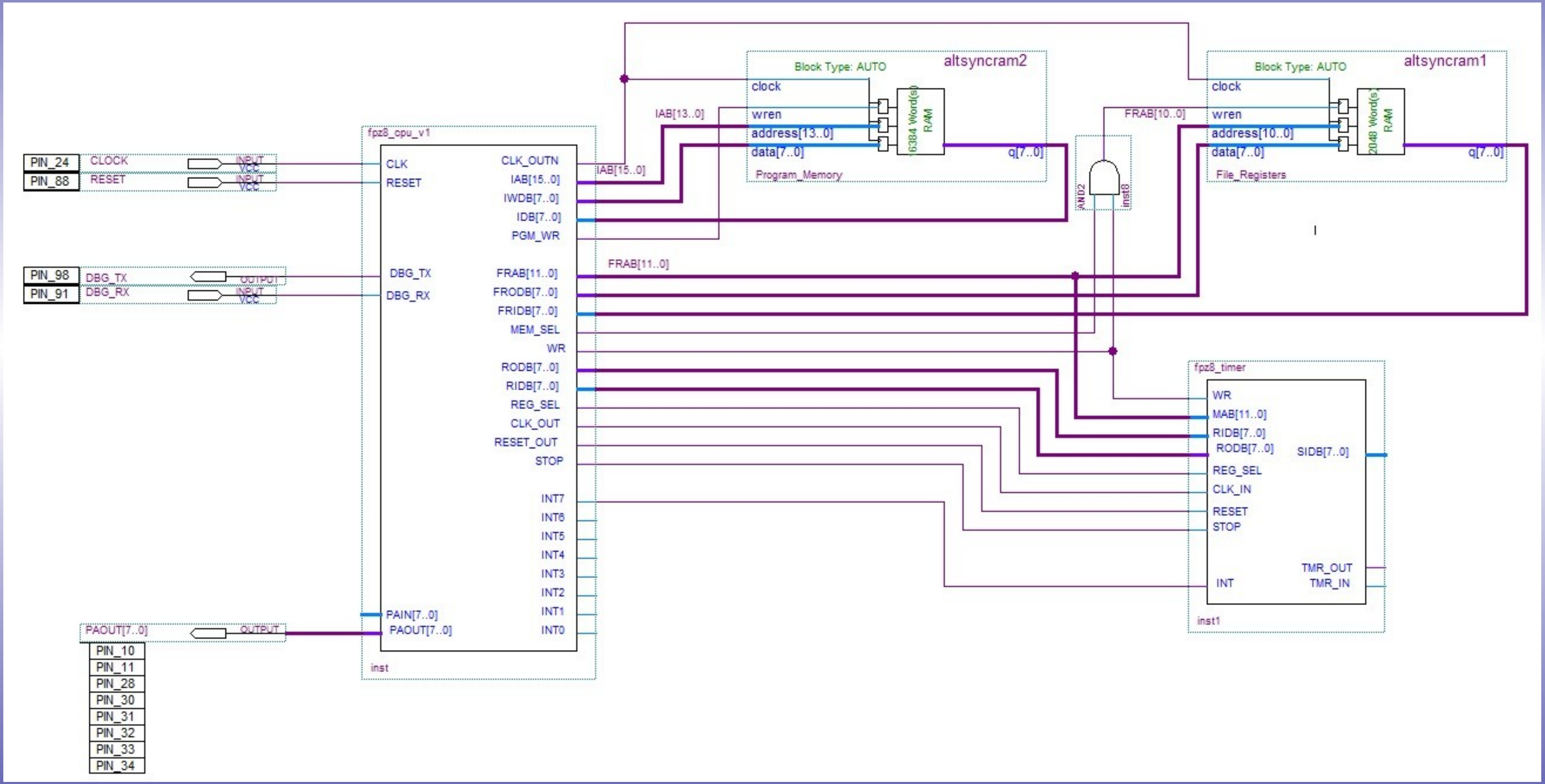
FPz8

- Softcore compatível com CPUs Z8 Encore (apenas algumas funcionalidades desnecessárias foram deixadas de fora, em especial as instruções LDE, LDEI e WDT)
- CPI não é igual ao do eZ8 original
- Sistema de interrupção com 8 vetores e prioridade configurável

FPz8

- 16Kb de memória de programa, 2Kb de RAM (configurável de acordo com os recursos internos do FPGA)
- On-Chip Debugger compatível com o OCD da Zilog
- Interface externa modular para periféricos (portas de E/S, timers, UART, etc)

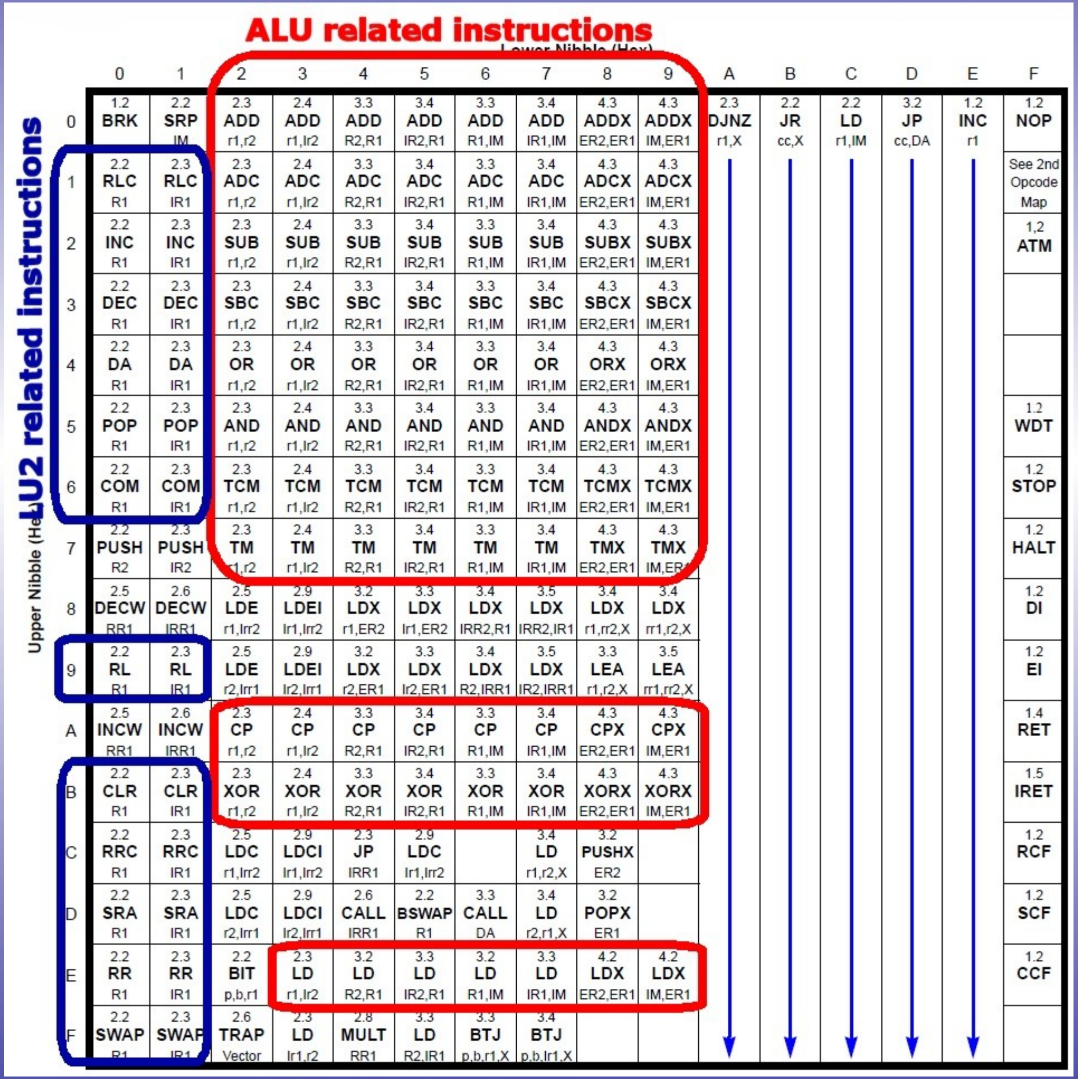
FPz8


altsyncram2
Block Type: AUTO
clock
wren
address[13..0]
data[7..0]
q[7..0]
16384 Word(s) RAM
Program_Memory
altsyncram1
Block Type: AUTO
clock
wren
address[10..0]
data[7..0]
q[7..0]
2048 Word(s) RAM
File_Registers
IAB[13..0]
IAB[15..0]
FRAB[10..0]
FRAB[11..0]
fpz8_cpu_v1
CLK
RESET
CLK_OUTN
IAB[15..0]
IWDB[7..0]
IDB[7..0]
PGM_WR
DBG_TX
DBG_RX
FRAB[11..0]
FRODB[7..0]
FRIDB[7..0]
MEM_SEL
WR
RODB[7..0]
RIDB[7..0]
REG_SEL
CLK_OUT
RESET_OUT
STOP
INT7
INT6
INT5
INT4
INT3
INT2
INT1
INT0
PAIN[7..0]
PAOUT[7..0]
inst
PIN_24
CLOCK
PIN_88
RESET
PIN_98
DBG_TX
PIN_91
DBG_RX
PAOUT[7..0]
PIN_10
PIN_11
PIN_28
PIN_30
PIN_31
PIN_32
PIN_33
PIN_34
AND2
inst8
fpz8_timer
WR
MAB[11..0]
RIDB[7..0]
RODB[7..0]
REG_SEL
CLK_IN
RESET
STOP
INT
SIDB[7..0]
TMR_OUT
TMR_IN
inst1

FPz8 - Barramentos

- IAB – Instruction Address Bus
- IWDB – Instruction Write Data Bus
- IDB – Instruction Data Bus
- FRAB – File Register Address Bus
- FRODB – File Register Output Data Bus
- FRIDB – File Register Input Data Bus
- RODB – Register Output Data Bus
- RIDB – Register Input Data Bus

FPz8


ALU related instructions

LU2 related instructions

Lower Nibble (Hex)

Upper Nibble (Hex)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	1.2 **BRK**	2.2 **SRP** IM	2.3 **ADD** r1,r2	2.4 **ADD** r1,Ir2	3.3 **ADD** R2,R1	3.4 **ADD** IR2,R1	3.3 **ADD** R1,IM	3.4 **ADD** IR1,IM	4.3 **ADDX** ER2,ER1	4.3 **ADDX** IM,ER1	2.3 **DJNZ** r1,X	2.2 **JR** cc,X	2.2 **LD** r1,IM	3.2 **JP** cc,DA	1.2 **INC** r1	1.2 **NOP**
1	2.2 **RLC** R1	2.3 **RLC** IR1	2.3 **ADC** r1,r2	2.4 **ADC** r1,Ir2	3.3 **ADC** R2,R1	3.4 **ADC** IR2,R1	3.3 **ADC** R1,IM	3.4 **ADC** IR1,IM	4.3 **ADCX** ER2,ER1	4.3 **ADCX** IM,ER1						See 2nd Opcode Map
2	2.2 **INC** R1	2.3 **INC** IR1	2.3 **SUB** r1,r2	2.4 **SUB** r1,Ir2	3.3 **SUB** R2,R1	3.4 **SUB** IR2,R1	3.3 **SUB** R1,IM	3.4 **SUB** IR1,IM	4.3 **SUBX** ER2,ER1	4.3 **SUBX** IM,ER1						1,2 **ATM**
3	2.2 **DEC** R1	2.3 **DEC** IR1	2.3 **SBC** r1,r2	2.4 **SBC** r1,Ir2	3.3 **SBC** R2,R1	3.4 **SBC** IR2,R1	3.3 **SBC** R1,IM	3.4 **SBC** IR1,IM	4.3 **SBCX** ER2,ER1	4.3 **SBCX** IM,ER1						
4	2.2 **DA** R1	2.3 **DA** IR1	2.3 **OR** r1,r2	2.4 **OR** r1,Ir2	3.3 **OR** R2,R1	3.4 **OR** IR2,R1	3.3 **OR** R1,IM	3.4 **OR** IR1,IM	4.3 **ORX** ER2,ER1	4.3 **ORX** IM,ER1						
5	2.2 **POP** R1	2.3 **POP** IR1	2.3 **AND** r1,r2	2.4 **AND** r1,Ir2	3.3 **AND** R2,R1	3.4 **AND** IR2,R1	3.3 **AND** R1,IM	3.4 **AND** IR1,IM	4.3 **ANDX** ER2,ER1	4.3 **ANDX** IM,ER1						1.2 **WDT**
6	2.2 **COM** R1	2.3 **COM** IR1	2.3 **TCM** r1,r2	2.4 **TCM** r1,Ir2	3.3 **TCM** R2,R1	3.4 **TCM** IR2,R1	3.3 **TCM** R1,IM	3.4 **TCM** IR1,IM	4.3 **TCMX** ER2,ER1	4.3 **TCMX** IM,ER1						1.2 **STOP**
7	2.2 **PUSH** R2	2.3 **PUSH** IR2	2.3 **TM** r1,r2	2.4 **TM** r1,Ir2	3.3 **TM** R2,R1	3.4 **TM** IR2,R1	3.3 **TM** R1,IM	3.4 **TM** IR1,IM	4.3 **TMX** ER2,ER1	4.3 **TMX** IM,ER1						1.2 **HALT**
8	2.5 **DECW** RR1	2.6 **DECW** IRR1	2.5 **LDE** r1,Irr2	2.9 **LDEI** Ir1,Irr2	3.2 **LDX** r1,ER2	3.3 **LDX** Ir1,ER2	3.4 **LDX** IRR2,R1	3.5 **LDX** IRR2,IR1	3.4 **LDX** r1,rr2,X	3.4 **LDX** rr1,r2,X						1.2 **DI**
9	2.2 **RL** R1	2.3 **RL** IR1	2.5 **LDE** r2,Irr1	2.9 **LDEI** Ir2,Irr1	3.2 **LDX** r2,ER1	3.3 **LDX** Ir2,ER1	3.4 **LDX** R2,IRR1	3.5 **LDX** IR2,IRR1	3.3 **LEA** r1,r2,X	3.5 **LEA** rr1,rr2,X						1.2 **EI**
A	2.5 **INCW** RR1	2.6 **INCW** IRR1	2.3 **CP** r1,r2	2.4 **CP** r1,Ir2	3.3 **CP** R2,R1	3.4 **CP** IR2,R1	3.3 **CP** R1,IM	3.4 **CP** IR1,IM	4.3 **CPX** ER2,ER1	4.3 **CPX** IM,ER1						1.4 **RET**
B	2.2 **CLR** R1	2.3 **CLR** IR1	2.3 **XOR** r1,r2	2.4 **XOR** r1,Ir2	3.3 **XOR** R2,R1	3.4 **XOR** IR2,R1	3.3 **XOR** R1,IM	3.4 **XOR** IR1,IM	4.3 **XORX** ER2,ER1	4.3 **XORX** IM,ER1						1.5 **IRET**
C	2.2 **RRC** R1	2.3 **RRC** IR1	2.5 **LDC** r1,Irr2	2.9 **LDCI** Ir1,Irr2	2.3 **JP** IRR1	2.9 **LDC** Ir1,Irr2		3.4 **LD** r1,r2,X	3.2 **PUSHX** ER2							1.2 **RCF**
D	2.2 **SRA** R1	2.3 **SRA** IR1	2.5 **LDC** r2,Irr1	2.9 **LDCI** Ir2,Irr1	2.6 **CALL** IRR1	2.2 **BSWAP** R1	3.3 **CALL** DA	3.4 **LD** r2,r1,X	3.2 **POPX** ER1							1.2 **SCF**
E	2.2 **RR** R1	2.3 **RR** IR1	2.2 **BIT** p,b,r1	2.3 **LD** r1,Ir2	3.2 **LD** R2,R1	3.3 **LD** IR2,R1	3.2 **LD** R1,IM	3.3 **LD** IR1,IM	4.2 **LDX** ER2,ER1	4.2 **LDX** IM,ER1						1.2 **CCF**
F	2.2 **SWAP** R1	2.3 **SWAP** IR1	2.6 **TRAP** Vector	2.3 **LD** Ir1,r2	2.8 **MULT** RR1	3.3 **LD** R2,IR1	3.3 **BTJ** p,b,r1,X	3.4 **BTJ** p,b,Ir1,X								

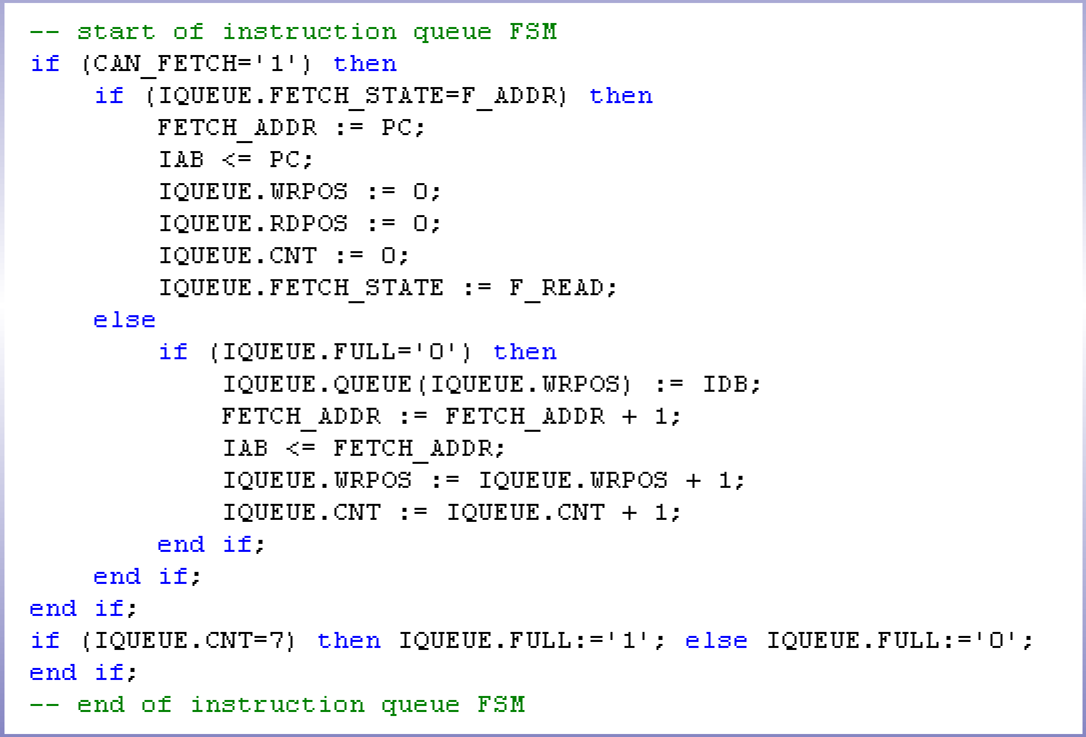
FPz8 - Design

- Instruction Queue (8 bytes)
- Decoder
- Interrupt Control
- Debugger

FPz8 – Instruction Queue

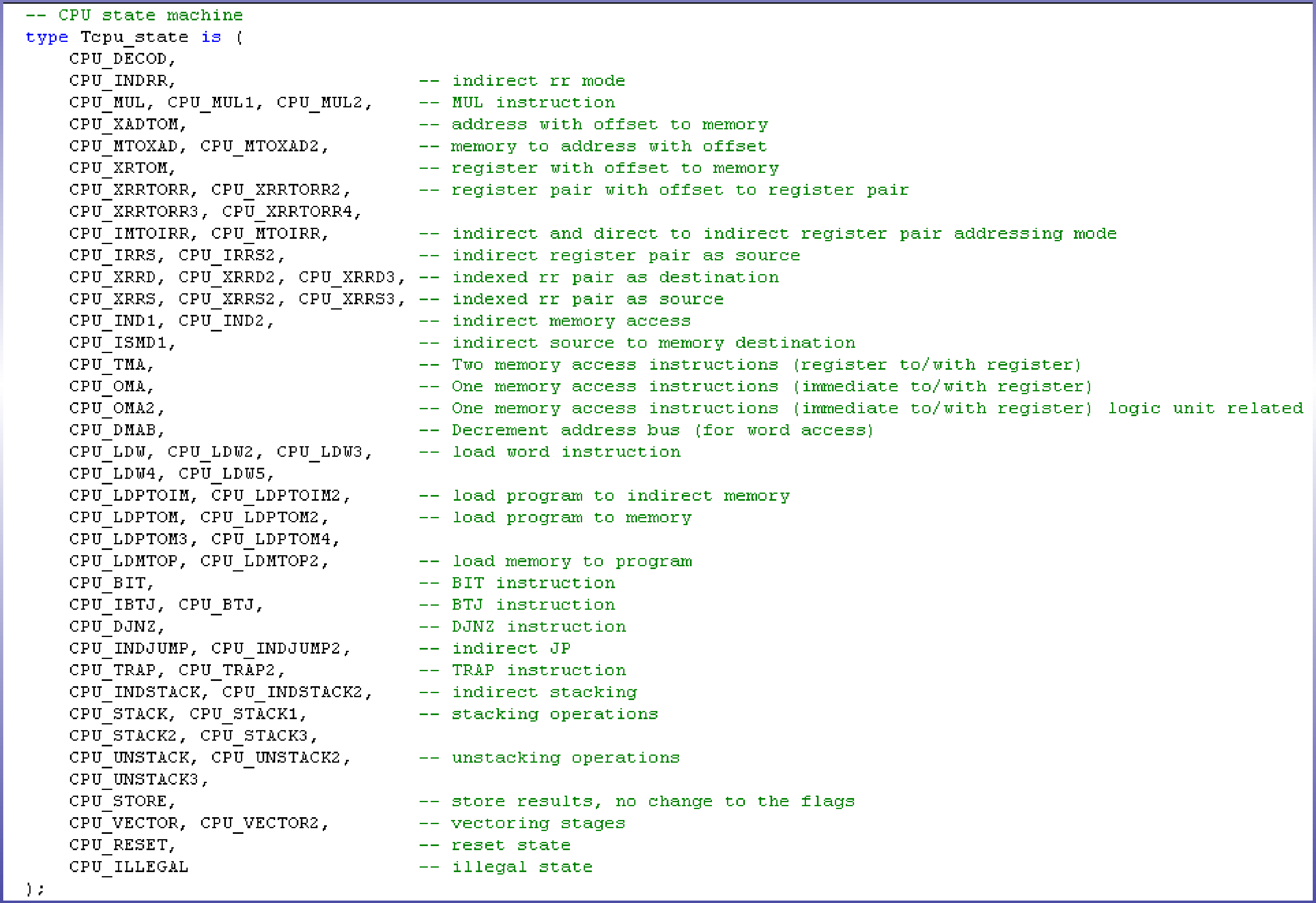
- Armazena até 8 bytes de opcodes lidos da memória de programa
- Dois ponteiros:
  - Escrita: controlado pela FSM de busca de opcodes
  - Leitura: controlado pelo decodificador de instruções
- Flush: instruções de desvio esvaziam a queue
- Stall: decodificador aguarda até que todos os bytes da instrução estejam disponíveis

FPz8 – Instruction Queue



```
-- start of instruction queue FSM
if (CAN_FETCH='1') then
    if (IQUEUE.FETCH_STATE=F_ADDR) then
        FETCH_ADDR := PC;
        IAB <= PC;
        IQUEUE.WRPOS := 0;
        IQUEUE.RDPOS := 0;
        IQUEUE.CNT := 0;
        IQUEUE.FETCH_STATE := F_READ;
    else
        if (IQUEUE.FULL='0') then
            IQUEUE.QUEUE(IQUEUE.WRPOS) := IDB;
            FETCH_ADDR := FETCH_ADDR + 1;
            IAB <= FETCH_ADDR;
            IQUEUE.WRPOS := IQUEUE.WRPOS + 1;
            IQUEUE.CNT := IQUEUE.CNT + 1;
        end if;
    end if;
end if;
if (IQUEUE.CNT=7) then IQUEUE.FULL:='1'; else IQUEUE.FULL:='0';
end if;
-- end of instruction queue FSM
```

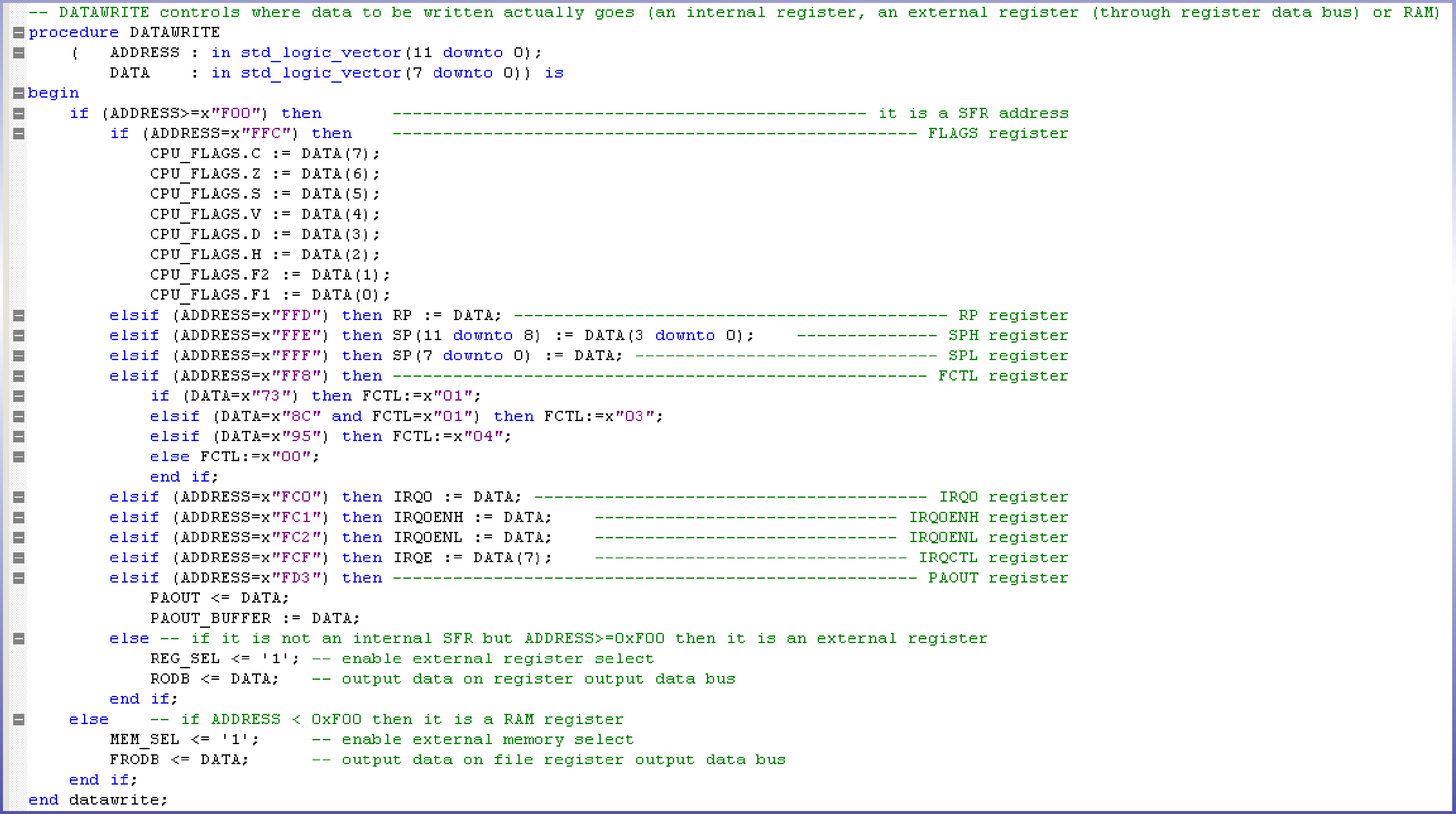
FPz8 - Decoder



```
-- CPU state machine
type Tcpu_state is (
    CPU_DECOD,
    CPU_INDRR,                        -- indirect rr mode
    CPU_MUL, CPU_MUL1, CPU_MUL2,      -- MUL instruction
    CPU_XADTOM,                       -- address with offset to memory
    CPU_MTOXAD, CPU_MTOXAD2,          -- memory to address with offset
    CPU_XRTOM,                        -- register with offset to memory
    CPU_XRRTORR, CPU_XRRTORR2,        -- register pair with offset to register pair
    CPU_XRRTORR3, CPU_XRRTORR4,
    CPU_IMTOIRR, CPU_MTOIRR,          -- indirect and direct to indirect register pair addressing mode
    CPU_IRRS, CPU_IRRS2,              -- indirect register pair as source
    CPU_XRRD, CPU_XRRD2, CPU_XRRD3,   -- indexed rr pair as destination
    CPU_XRRS, CPU_XRRS2, CPU_XRRS3,   -- indexed rr pair as source
    CPU_IND1, CPU_IND2,               -- indirect memory access
    CPU_ISMD1,                        -- indirect source to memory destination
    CPU_TMA,                          -- Two memory access instructions (register to/with register)
    CPU_OMA,                          -- One memory access instructions (immediate to/with register)
    CPU_OMA2,                         -- One memory access instructions (immediate to/with register) logic unit related
    CPU_DMAB,                         -- Decrement address bus (for word access)
    CPU_LDW, CPU_LDW2, CPU_LDW3,      -- load word instruction
    CPU_LDW4, CPU_LDW5,
    CPU_LDPTOIM, CPU_LDPTOIM2,        -- load program to indirect memory
    CPU_LDPTOM, CPU_LDPTOM2,          -- load program to memory
    CPU_LDPTOM3, CPU_LDPTOM4,
    CPU_LDMTOP, CPU_LDMTOP2,          -- load memory to program
    CPU_BIT,                          -- BIT instruction
    CPU_IBTJ, CPU_BTJ,                -- BTJ instruction
    CPU_DJNZ,                         -- DJNZ instruction
    CPU_INDJUMP, CPU_INDJUMP2,        -- indirect JP
    CPU_TRAP, CPU_TRAP2,              -- TRAP instruction
    CPU_INDSTACK, CPU_INDSTACK2,      -- indirect stacking
    CPU_STACK, CPU_STACK1,            -- stacking operations
    CPU_STACK2, CPU_STACK3,
    CPU_UNSTACK, CPU_UNSTACK2,        -- unstacking operations
    CPU_UNSTACK3,
    CPU_STORE,                        -- store results, no change to the flags
    CPU_VECTOR, CPU_VECTOR2,          -- vectoring stages
    CPU_RESET,                        -- reset state
    CPU_ILLEGAL                       -- illegal state
);
```

FPz8 - Decoder

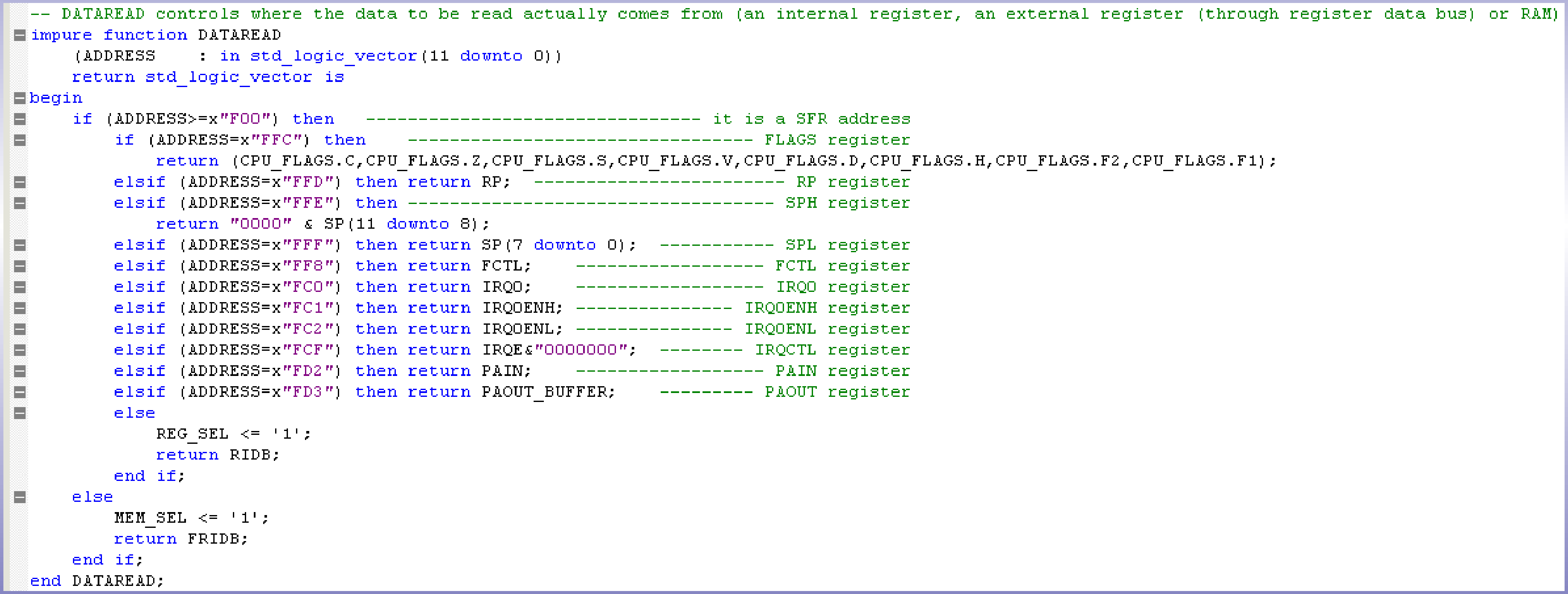
DATAWRITE



```vhdl
-- DATAWRITE controls where data to be written actually goes (an internal register, an external register (through register data bus) or RAM)
procedure DATAWRITE
    (   ADDRESS : in std_logic_vector(11 downto 0);
        DATA    : in std_logic_vector(7 downto 0)) is
begin
    if (ADDRESS>=x"F00") then      ----------------------------------------------- it is a SFR address
        if (ADDRESS=x"FFC") then   ---------------------------------------------------- FLAGS register
            CPU_FLAGS.C := DATA(7);
            CPU_FLAGS.Z := DATA(6);
            CPU_FLAGS.S := DATA(5);
            CPU_FLAGS.V := DATA(4);
            CPU_FLAGS.D := DATA(3);
            CPU_FLAGS.H := DATA(2);
            CPU_FLAGS.F2 := DATA(1);
            CPU_FLAGS.F1 := DATA(0);
        elsif (ADDRESS=x"FFD") then RP := DATA; ------------------------------------------- RP register
        elsif (ADDRESS=x"FFE") then SP(11 downto 8) := DATA(3 downto 0);    -------------- SPH register
        elsif (ADDRESS=x"FFF") then SP(7 downto 0) := DATA; ------------------------------ SPL register
        elsif (ADDRESS=x"FF8") then ----------------------------------------------------- FCTL register
            if (DATA=x"73") then FCTL:=x"01";
            elsif (DATA=x"8C" and FCTL=x"01") then FCTL:=x"03";
            elsif (DATA=x"95") then FCTL:=x"04";
            else FCTL:=x"00";
            end if;
        elsif (ADDRESS=x"FC0") then IRQ0 := DATA; -------------------------------------- IRQ0 register
        elsif (ADDRESS=x"FC1") then IRQ0ENH := DATA;    ------------------------------ IRQ0ENH register
        elsif (ADDRESS=x"FC2") then IRQ0ENL := DATA;    ------------------------------ IRQ0ENL register
        elsif (ADDRESS=x"FCF") then IRQE := DATA(7);    ------------------------------- IRQCTL register
        elsif (ADDRESS=x"FD3") then ---------------------------------------------------- PAOUT register
            PAOUT <= DATA;
            PAOUT_BUFFER := DATA;
        else -- if it is not an internal SFR but ADDRESS>=0xF00 then it is an external register
            REG_SEL <= '1'; -- enable external register select
            RODB <= DATA;   -- output data on register output data bus
        end if;
    else     -- if ADDRESS < 0xF00 then it is a RAM register
        MEM_SEL <= '1';     -- enable external memory select
        FRODB <= DATA;      -- output data on file register output data bus
    end if;
end datawrite;
```

FPz8 - Decoder

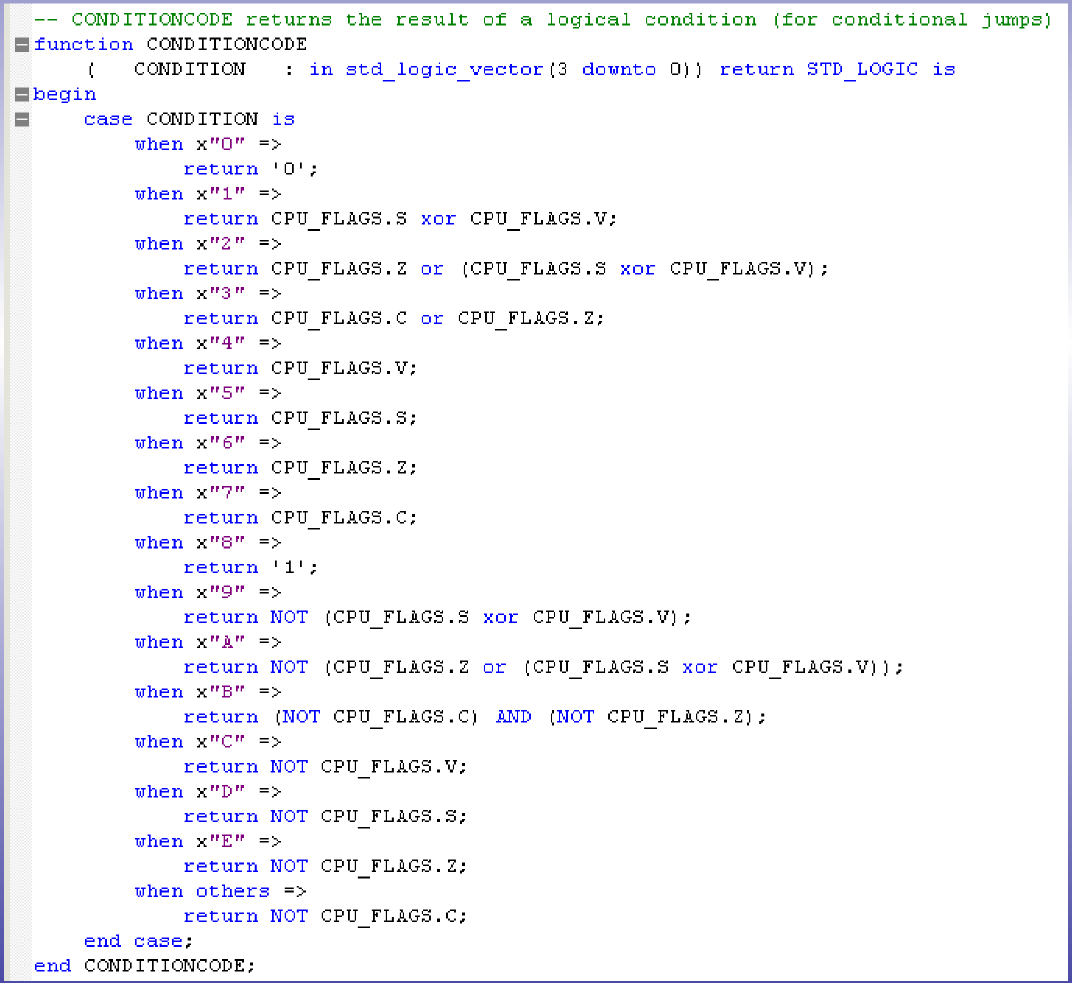
DATAREAD



```
-- DATAREAD controls where the data to be read actually comes from (an internal register, an external register (through register data bus) or RAM)
impure function DATAREAD
    (ADDRESS    : in std_logic_vector(11 downto 0))
    return std_logic_vector is
begin
    if (ADDRESS>=x"F00") then   -------------------------------- it is a SFR address
        if (ADDRESS=x"FFC") then    --------------------------------- FLAGS register
            return (CPU_FLAGS.C,CPU_FLAGS.Z,CPU_FLAGS.S,CPU_FLAGS.V,CPU_FLAGS.D,CPU_FLAGS.H,CPU_FLAGS.F2,CPU_FLAGS.F1);
        elsif (ADDRESS=x"FFD") then return RP;  ------------------------ RP register
        elsif (ADDRESS=x"FFE") then ----------------------------------- SPH register
            return "0000" & SP(11 downto 8);
        elsif (ADDRESS=x"FFF") then return SP(7 downto 0);  ----------- SPL register
        elsif (ADDRESS=x"FF8") then return FCTL;    ------------------ FCTL register
        elsif (ADDRESS=x"FC0") then return IRQ0;    ------------------ IRQ0 register
        elsif (ADDRESS=x"FC1") then return IRQ0ENH; --------------- IRQ0ENH register
        elsif (ADDRESS=x"FC2") then return IRQ0ENL; --------------- IRQ0ENL register
        elsif (ADDRESS=x"FCF") then return IRQE&"0000000";  -------- IRQCTL register
        elsif (ADDRESS=x"FD2") then return PAIN;    ------------------ PAIN register
        elsif (ADDRESS=x"FD3") then return PAOUT_BUFFER;    --------- PAOUT register
        else
            REG_SEL <= '1';
            return RIDB;
        end if;
    else
        MEM_SEL <= '1';
        return FRIDB;
    end if;
end DATAREAD;
```

FPz8 - Decoder

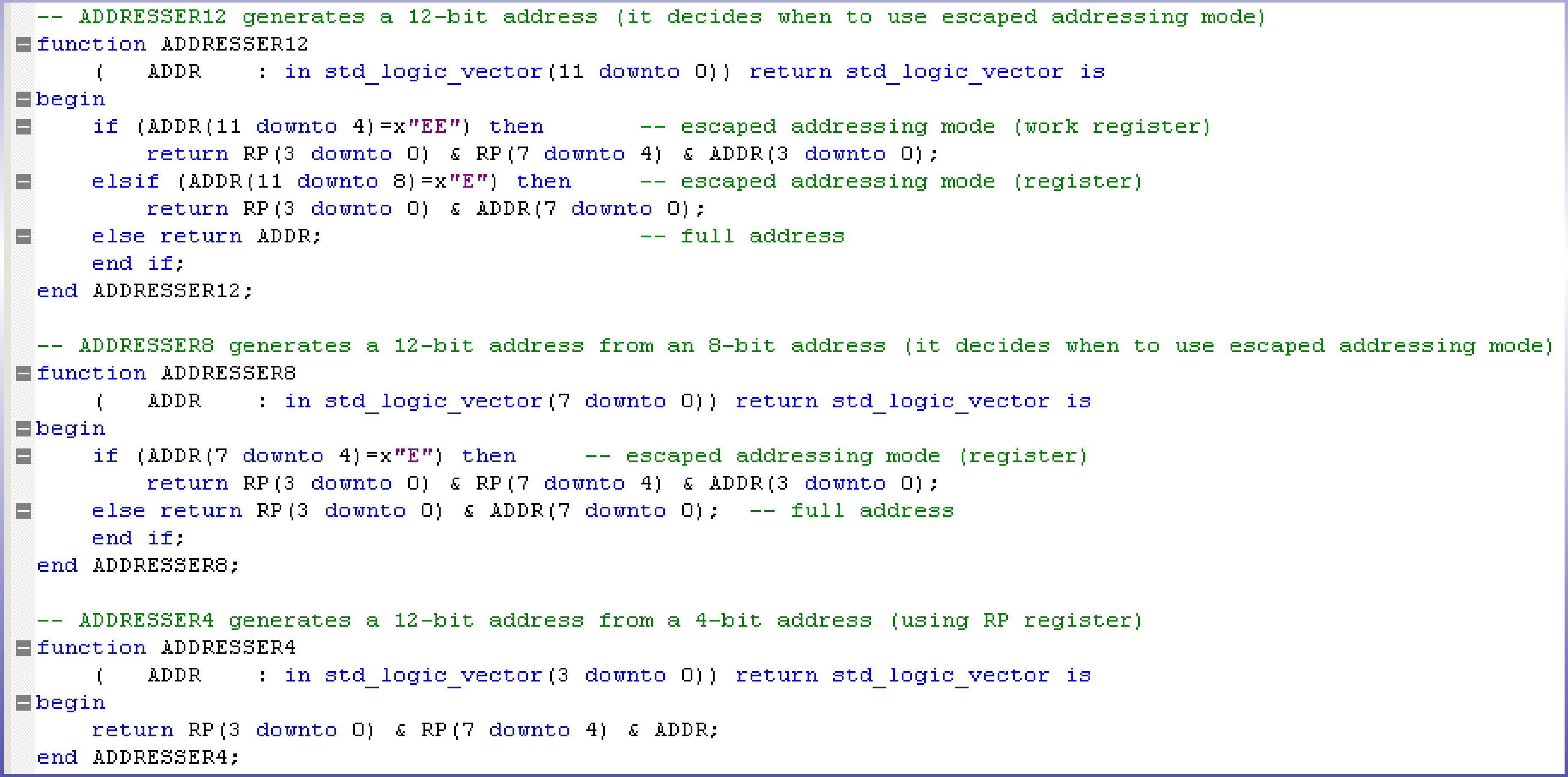
CONDITIONCODE



```
-- CONDITIONCODE returns the result of a logical condition (for conditional jumps)
function CONDITIONCODE
    (   CONDITION   : in std_logic_vector(3 downto 0)) return STD_LOGIC is
begin
    case CONDITION is
        when x"0" =>
            return '0';
        when x"1" =>
            return CPU_FLAGS.S xor CPU_FLAGS.V;
        when x"2" =>
            return CPU_FLAGS.Z or (CPU_FLAGS.S xor CPU_FLAGS.V);
        when x"3" =>
            return CPU_FLAGS.C or CPU_FLAGS.Z;
        when x"4" =>
            return CPU_FLAGS.V;
        when x"5" =>
            return CPU_FLAGS.S;
        when x"6" =>
            return CPU_FLAGS.Z;
        when x"7" =>
            return CPU_FLAGS.C;
        when x"8" =>
            return '1';
        when x"9" =>
            return NOT (CPU_FLAGS.S xor CPU_FLAGS.V);
        when x"A" =>
            return NOT (CPU_FLAGS.Z or (CPU_FLAGS.S xor CPU_FLAGS.V));
        when x"B" =>
            return (NOT CPU_FLAGS.C) AND (NOT CPU_FLAGS.Z);
        when x"C" =>
            return NOT CPU_FLAGS.V;
        when x"D" =>
            return NOT CPU_FLAGS.S;
        when x"E" =>
            return NOT CPU_FLAGS.Z;
        when others =>
            return NOT CPU_FLAGS.C;
    end case;
end CONDITIONCODE;
```

FPz8 - Decoder

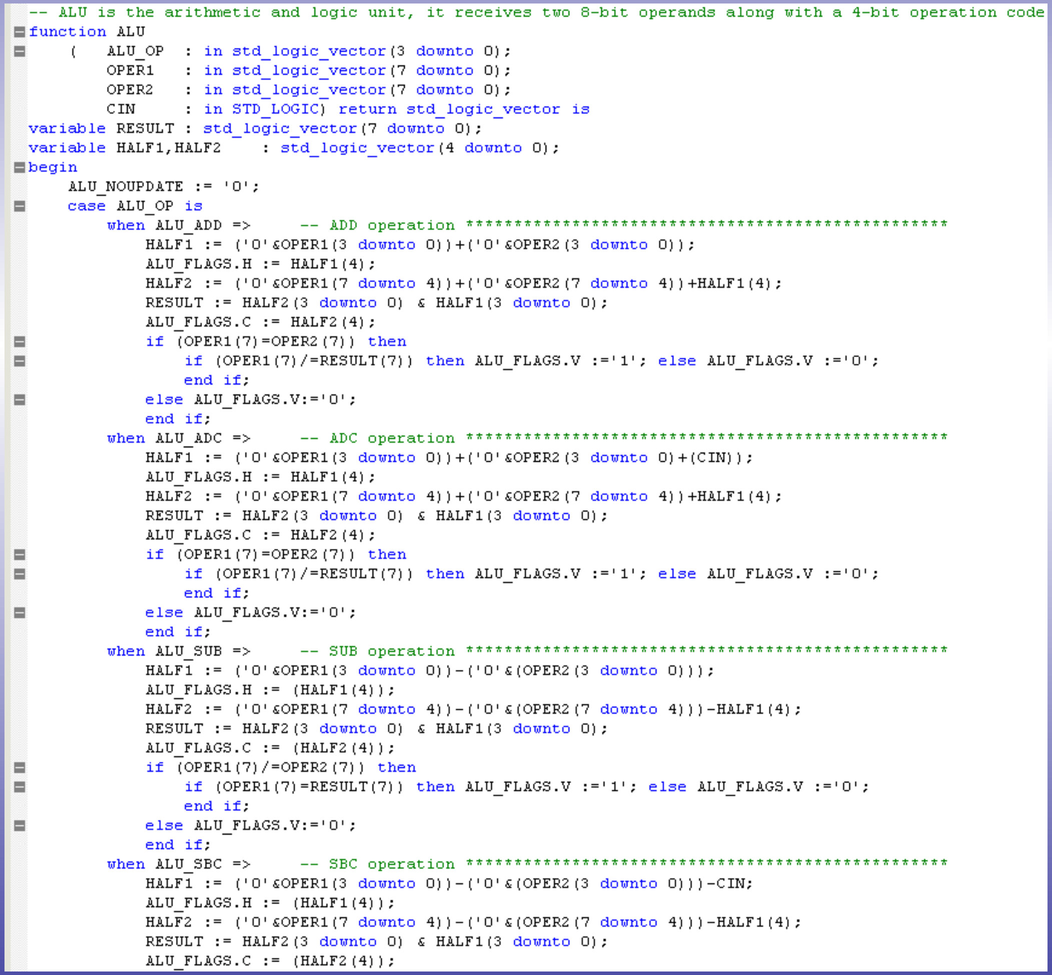
ADDRESSER12 / ADDRESSER8 / ADDRESSER4



```
-- ADDRESSER12 generates a 12-bit address (it decides when to use escaped addressing mode)
function ADDRESSER12
    (   ADDR    : in std_logic_vector(11 downto 0)) return std_logic_vector is
begin
    if (ADDR(11 downto 4)=x"EE") then       -- escaped addressing mode (work register)
        return RP(3 downto 0) & RP(7 downto 4) & ADDR(3 downto 0);
    elsif (ADDR(11 downto 8)=x"E") then     -- escaped addressing mode (register)
        return RP(3 downto 0) & ADDR(7 downto 0);
    else return ADDR;                       -- full address
    end if;
end ADDRESSER12;

-- ADDRESSER8 generates a 12-bit address from an 8-bit address (it decides when to use escaped addressing mode)
function ADDRESSER8
    (   ADDR    : in std_logic_vector(7 downto 0)) return std_logic_vector is
begin
    if (ADDR(7 downto 4)=x"E") then     -- escaped addressing mode (register)
        return RP(3 downto 0) & RP(7 downto 4) & ADDR(3 downto 0);
    else return RP(3 downto 0) & ADDR(7 downto 0);  -- full address
    end if;
end ADDRESSER8;

-- ADDRESSER4 generates a 12-bit address from a 4-bit address (using RP register)
function ADDRESSER4
    (   ADDR    : in std_logic_vector(3 downto 0)) return std_logic_vector is
begin
    return RP(3 downto 0) & RP(7 downto 4) & ADDR;
end ADDRESSER4;
```

FPz8 - Decoder

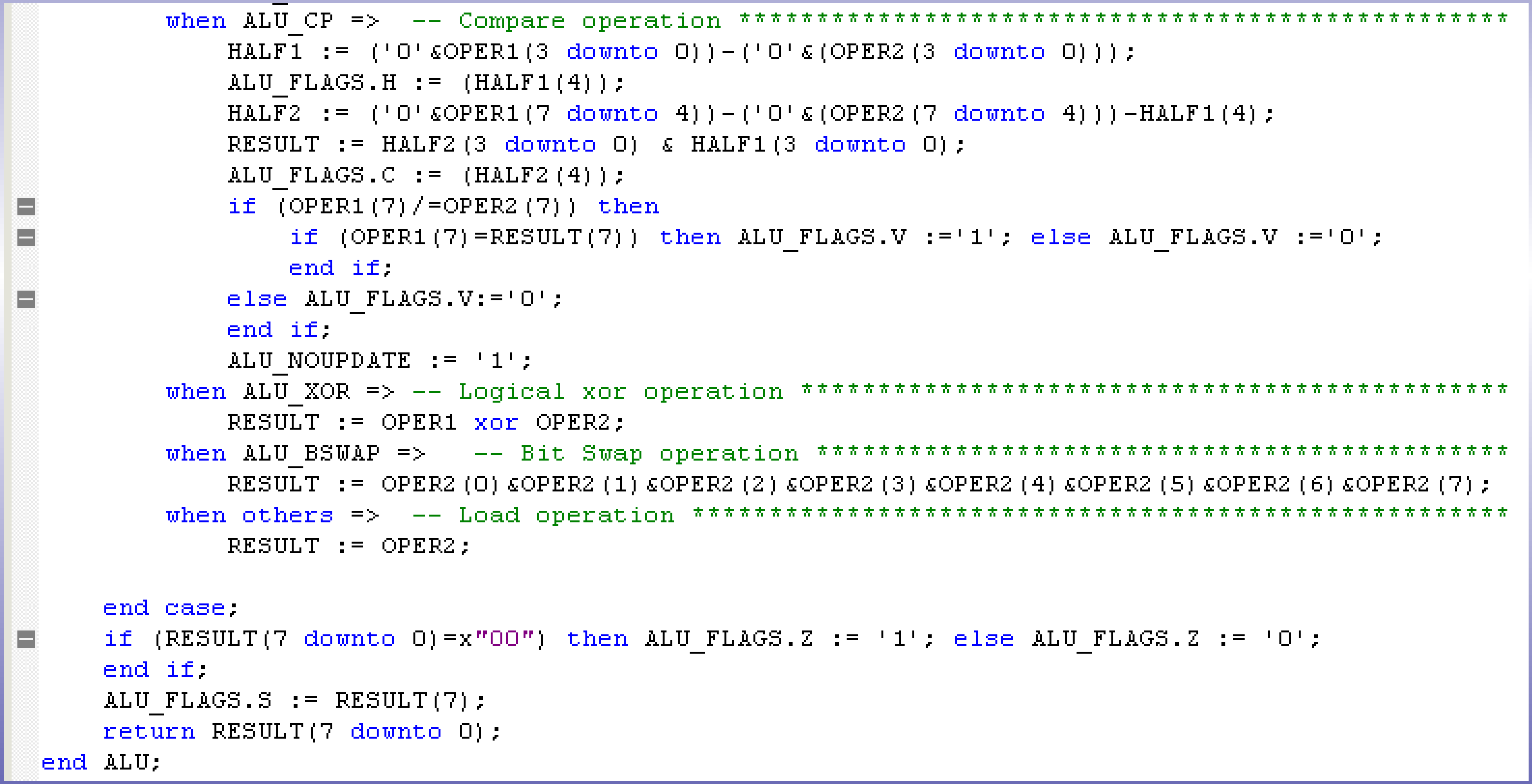
ALU



```vhdl
-- ALU is the arithmetic and logic unit, it receives two 8-bit operands along with a 4-bit operation code
function ALU
    (   ALU_OP   : in std_logic_vector(3 downto 0);
        OPER1    : in std_logic_vector(7 downto 0);
        OPER2    : in std_logic_vector(7 downto 0);
        CIN      : in STD_LOGIC) return std_logic_vector is
variable RESULT : std_logic_vector(7 downto 0);
variable HALF1,HALF2    : std_logic_vector(4 downto 0);
begin
    ALU_NOUPDATE := '0';
    case ALU_OP is
        when ALU_ADD =>     -- ADD operation **************************************************
            HALF1 := ('0'&OPER1(3 downto 0))+('0'&OPER2(3 downto 0));
            ALU_FLAGS.H := HALF1(4);
            HALF2 := ('0'&OPER1(7 downto 4))+('0'&OPER2(7 downto 4))+HALF1(4);
            RESULT := HALF2(3 downto 0) & HALF1(3 downto 0);
            ALU_FLAGS.C := HALF2(4);
            if (OPER1(7)=OPER2(7)) then
                if (OPER1(7)/=RESULT(7)) then ALU_FLAGS.V :='1'; else ALU_FLAGS.V :='0';
                end if;
            else ALU_FLAGS.V:='0';
            end if;
        when ALU_ADC =>     -- ADC operation **************************************************
            HALF1 := ('0'&OPER1(3 downto 0))+('0'&OPER2(3 downto 0)+(CIN));
            ALU_FLAGS.H := HALF1(4);
            HALF2 := ('0'&OPER1(7 downto 4))+('0'&OPER2(7 downto 4))+HALF1(4);
            RESULT := HALF2(3 downto 0) & HALF1(3 downto 0);
            ALU_FLAGS.C := HALF2(4);
            if (OPER1(7)=OPER2(7)) then
                if (OPER1(7)/=RESULT(7)) then ALU_FLAGS.V :='1'; else ALU_FLAGS.V :='0';
                end if;
            else ALU_FLAGS.V:='0';
            end if;
        when ALU_SUB =>     -- SUB operation **************************************************
            HALF1 := ('0'&OPER1(3 downto 0))-('0'&(OPER2(3 downto 0)));
            ALU_FLAGS.H := (HALF1(4));
            HALF2 := ('0'&OPER1(7 downto 4))-('0'&(OPER2(7 downto 4)))-HALF1(4);
            RESULT := HALF2(3 downto 0) & HALF1(3 downto 0);
            ALU_FLAGS.C := (HALF2(4));
            if (OPER1(7)/=OPER2(7)) then
                if (OPER1(7)=RESULT(7)) then ALU_FLAGS.V :='1'; else ALU_FLAGS.V :='0';
                end if;
            else ALU_FLAGS.V:='0';
            end if;
        when ALU_SBC =>     -- SBC operation **************************************************
            HALF1 := ('0'&OPER1(3 downto 0))-('0'&(OPER2(3 downto 0)))-CIN;
            ALU_FLAGS.H := (HALF1(4));
            HALF2 := ('0'&OPER1(7 downto 4))-('0'&(OPER2(7 downto 4)))-HALF1(4);
            RESULT := HALF2(3 downto 0) & HALF1(3 downto 0);
            ALU_FLAGS.C := (HALF2(4));
```

FPz8 - Decoder

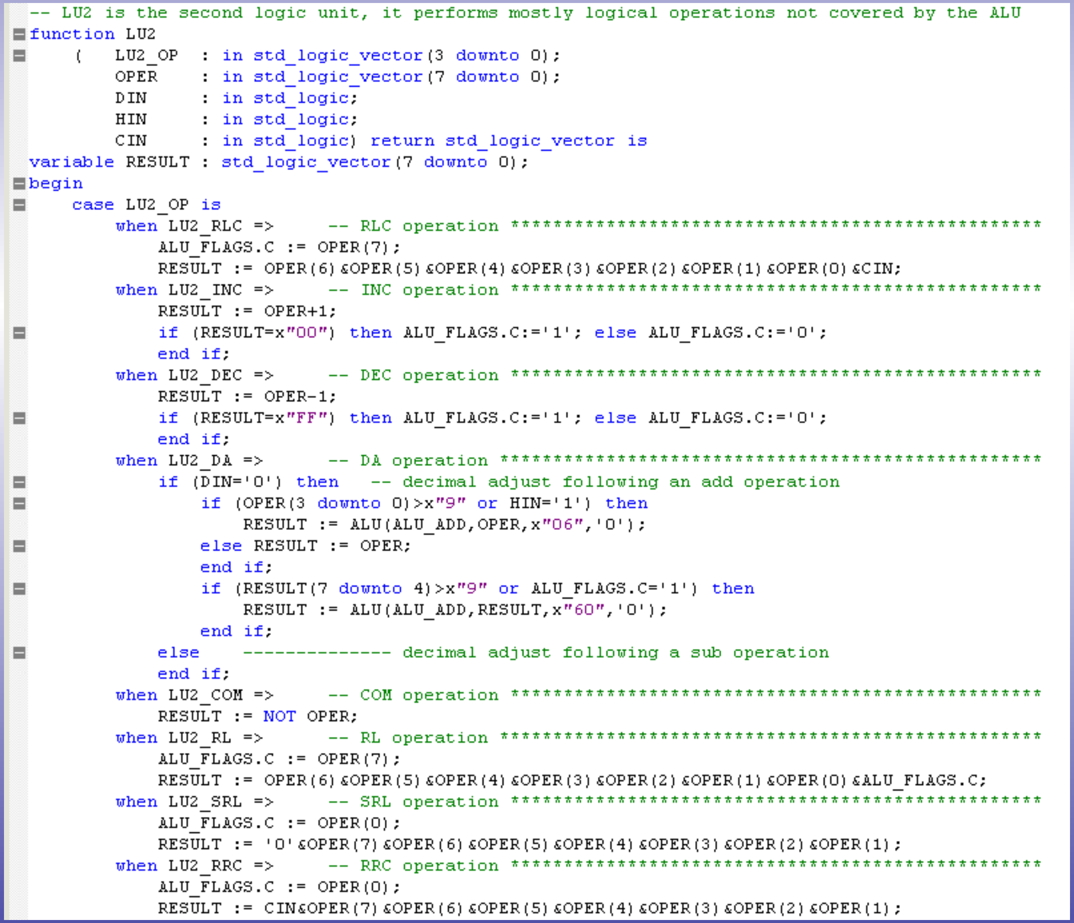
ALU



```vhdl
        when ALU_CP =>  -- Compare operation ***************************************************
            HALF1 := ('0'&OPER1(3 downto 0))-('0'&(OPER2(3 downto 0)));
            ALU_FLAGS.H := (HALF1(4));
            HALF2 := ('0'&OPER1(7 downto 4))-('0'&(OPER2(7 downto 4)))-HALF1(4);
            RESULT := HALF2(3 downto 0) & HALF1(3 downto 0);
            ALU_FLAGS.C := (HALF2(4));
            if (OPER1(7)/=OPER2(7)) then
                if (OPER1(7)=RESULT(7)) then ALU_FLAGS.V :='1'; else ALU_FLAGS.V :='0';
                end if;
            else ALU_FLAGS.V:='0';
            end if;
            ALU_NOUPDATE := '1';
        when ALU_XOR => -- Logical xor operation ********************************************
            RESULT := OPER1 xor OPER2;
        when ALU_BSWAP =>   -- Bit Swap operation *******************************************
            RESULT := OPER2(0)&OPER2(1)&OPER2(2)&OPER2(3)&OPER2(4)&OPER2(5)&OPER2(6)&OPER2(7);
        when others =>  -- Load operation ***************************************************
            RESULT := OPER2;

    end case;
    if (RESULT(7 downto 0)=x"00") then ALU_FLAGS.Z := '1'; else ALU_FLAGS.Z := '0';
    end if;
    ALU_FLAGS.S := RESULT(7);
    return RESULT(7 downto 0);
end ALU;
```

FPz8 - Decoder

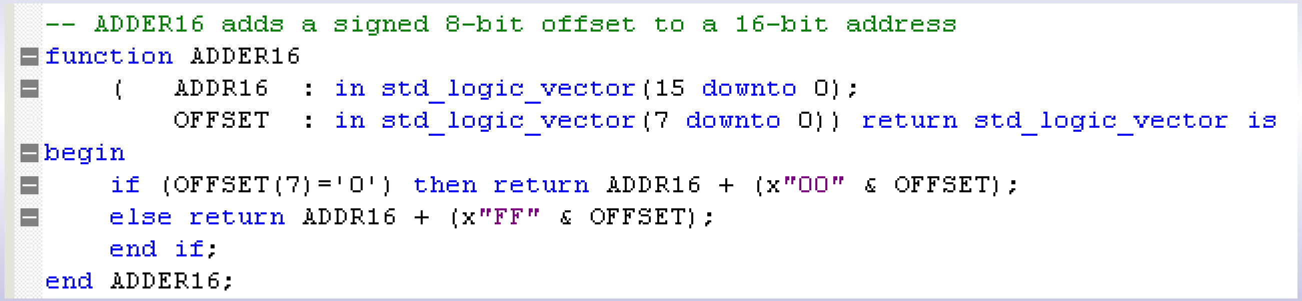
LU2



```vhdl
-- LU2 is the second logic unit, it performs mostly logical operations not covered by the ALU
function LU2
    (   LU2_OP  : in std_logic_vector(3 downto 0);
        OPER    : in std_logic_vector(7 downto 0);
        DIN     : in std_logic;
        HIN     : in std_logic;
        CIN     : in std_logic) return std_logic_vector is
variable RESULT : std_logic_vector(7 downto 0);
begin
    case LU2_OP is
        when LU2_RLC =>     -- RLC operation **************************************************
            ALU_FLAGS.C := OPER(7);
            RESULT := OPER(6)&OPER(5)&OPER(4)&OPER(3)&OPER(2)&OPER(1)&OPER(0)&CIN;
        when LU2_INC =>     -- INC operation **************************************************
            RESULT := OPER+1;
            if (RESULT=x"00") then ALU_FLAGS.C:='1'; else ALU_FLAGS.C:='0';
            end if;
        when LU2_DEC =>     -- DEC operation **************************************************
            RESULT := OPER-1;
            if (RESULT=x"FF") then ALU_FLAGS.C:='1'; else ALU_FLAGS.C:='0';
            end if;
        when LU2_DA =>      -- DA operation ***************************************************
            if (DIN='0') then   -- decimal adjust following an add operation
                if (OPER(3 downto 0)>x"9" or HIN='1') then
                    RESULT := ALU(ALU_ADD,OPER,x"06",'0');
                else RESULT := OPER;
                end if;
                if (RESULT(7 downto 4)>x"9" or ALU_FLAGS.C='1') then
                    RESULT := ALU(ALU_ADD,RESULT,x"60",'0');
                end if;
            else    -------------- decimal adjust following a sub operation
            end if;
        when LU2_COM =>     -- COM operation **************************************************
            RESULT := NOT OPER;
        when LU2_RL =>      -- RL operation ***************************************************
            ALU_FLAGS.C := OPER(7);
            RESULT := OPER(6)&OPER(5)&OPER(4)&OPER(3)&OPER(2)&OPER(1)&OPER(0)&ALU_FLAGS.C;
        when LU2_SRL =>     -- SRL operation **************************************************
            ALU_FLAGS.C := OPER(0);
            RESULT := '0'&OPER(7)&OPER(6)&OPER(5)&OPER(4)&OPER(3)&OPER(2)&OPER(1);
        when LU2_RRC =>     -- RRC operation **************************************************
            ALU_FLAGS.C := OPER(0);
            RESULT := CIN&OPER(7)&OPER(6)&OPER(5)&OPER(4)&OPER(3)&OPER(2)&OPER(1);
```

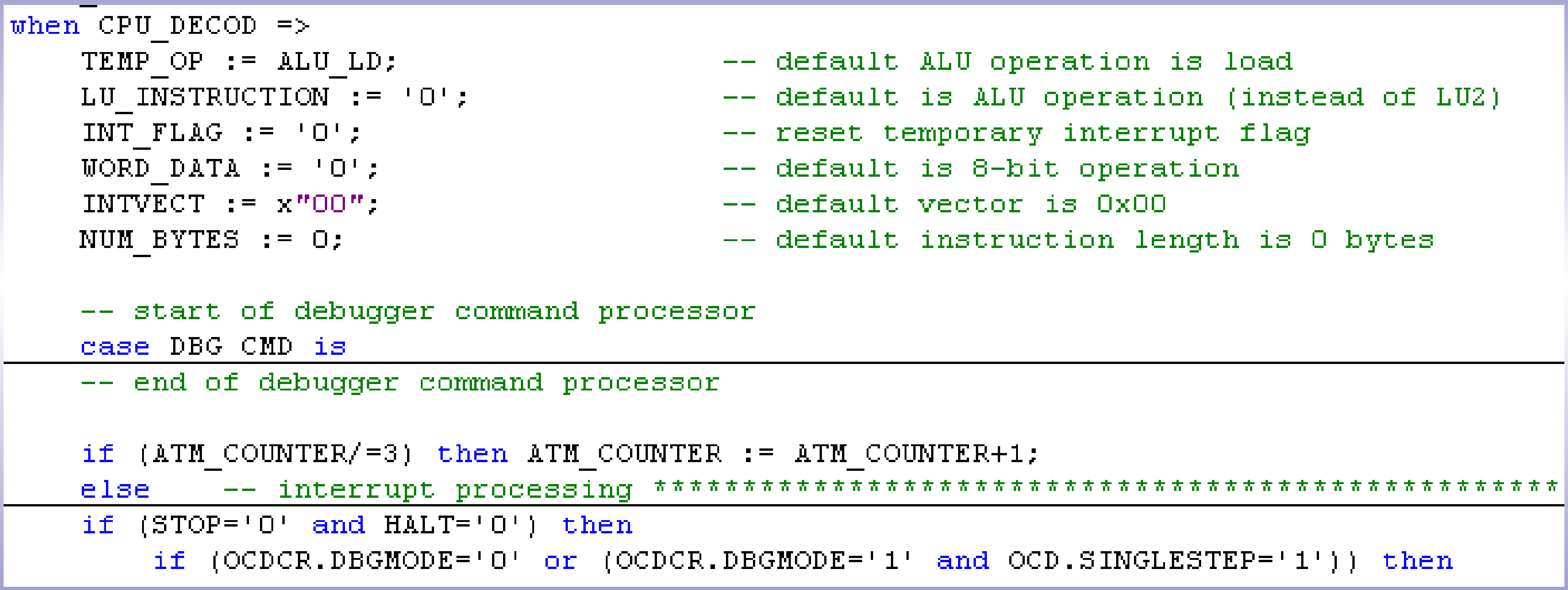
FPz8 - Decoder

ADDER16



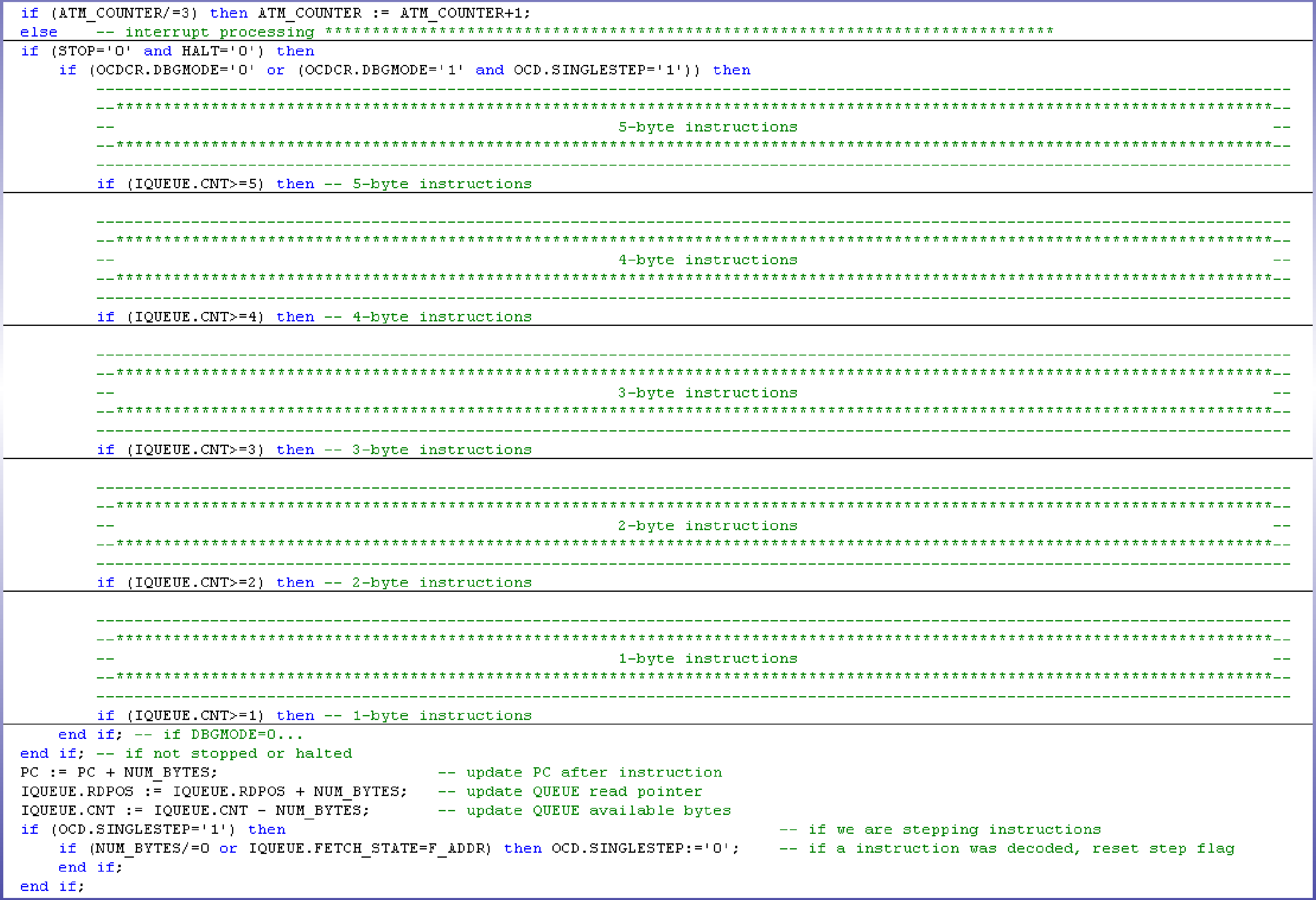
```
-- ADDER16 adds a signed 8-bit offset to a 16-bit address
function ADDER16
    (   ADDR16  : in std_logic_vector(15 downto 0);
        OFFSET  : in std_logic_vector(7 downto 0)) return std_logic_vector is
begin
    if (OFFSET(7)='0') then return ADDR16 + (x"00" & OFFSET);
    else return ADDR16 + (x"FF" & OFFSET);
    end if;
end ADDER16;
```

FPz8 - Decoder



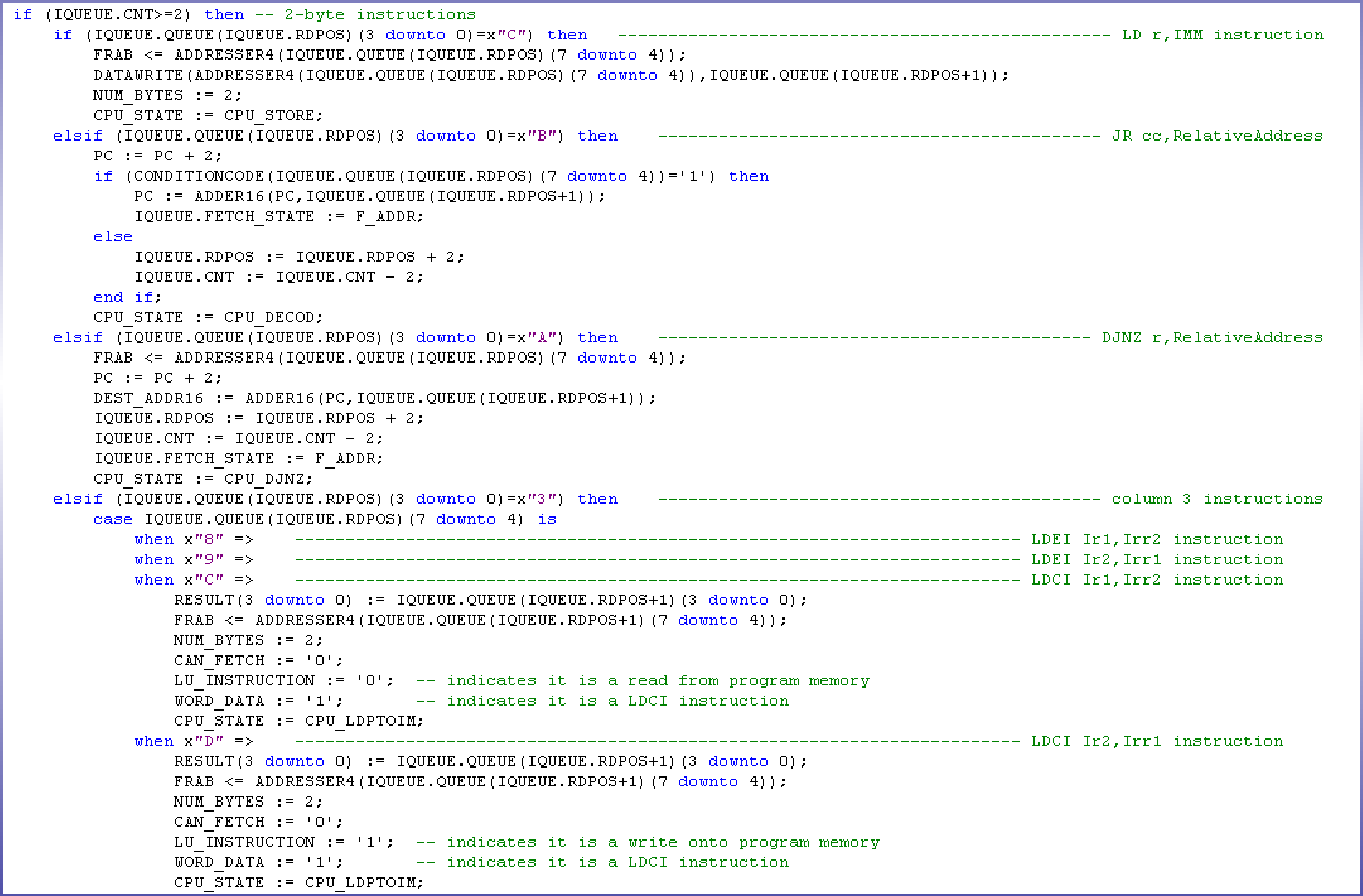
```
when CPU_DECOD =>
    TEMP_OP := ALU_LD;                  -- default ALU operation is load
    LU_INSTRUCTION := '0';              -- default is ALU operation (instead of LU2)
    INT_FLAG := '0';                    -- reset temporary interrupt flag
    WORD_DATA := '0';                   -- default is 8-bit operation
    INTVECT := x"00";                   -- default vector is 0x00
    NUM_BYTES := 0;                     -- default instruction length is 0 bytes

    -- start of debugger command processor
    case DBG_CMD is
    -- end of debugger command processor

    if (ATM_COUNTER/=3) then ATM_COUNTER := ATM_COUNTER+1;
    else    -- interrupt processing ***************************************************
    if (STOP='0' and HALT='0') then
        if (OCDCR.DBGMODE='0' or (OCDCR.DBGMODE='1' and OCD.SINGLESTEP='1')) then
```

FPz8 - Decoder



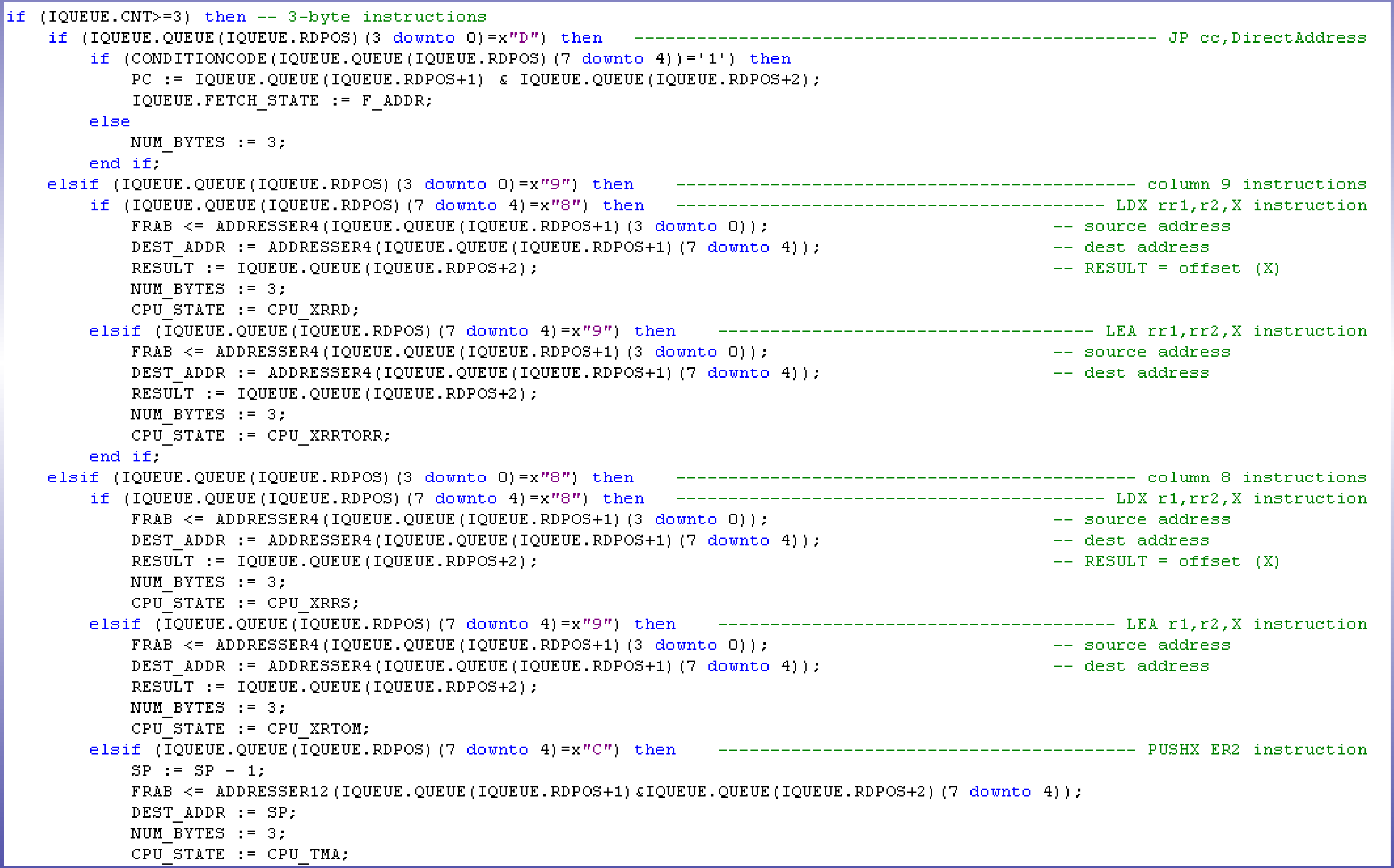
```
if (ATM_COUNTER/=3) then ATM_COUNTER := ATM_COUNTER+1;
else    -- interrupt processing *******************************************************************************
if (STOP='0' and HALT='0') then
    if (OCDCR.DBGMODE='0' or (OCDCR.DBGMODE='1' and OCD.SINGLESTEP='1')) then
        ---------------------------------------------------------------------------------------------------------------------------------
        --*****************************************************************************************************************************--
        --                                                      5-byte instructions                                                    --
        --*****************************************************************************************************************************--
        ---------------------------------------------------------------------------------------------------------------------------------
        if (IQUEUE.CNT>=5) then -- 5-byte instructions

        ---------------------------------------------------------------------------------------------------------------------------------
        --*****************************************************************************************************************************--
        --                                                      4-byte instructions                                                    --
        --*****************************************************************************************************************************--
        ---------------------------------------------------------------------------------------------------------------------------------
        if (IQUEUE.CNT>=4) then -- 4-byte instructions

        ---------------------------------------------------------------------------------------------------------------------------------
        --*****************************************************************************************************************************--
        --                                                      3-byte instructions                                                    --
        --*****************************************************************************************************************************--
        ---------------------------------------------------------------------------------------------------------------------------------
        if (IQUEUE.CNT>=3) then -- 3-byte instructions

        ---------------------------------------------------------------------------------------------------------------------------------
        --*****************************************************************************************************************************--
        --                                                      2-byte instructions                                                    --
        --*****************************************************************************************************************************--
        ---------------------------------------------------------------------------------------------------------------------------------
        if (IQUEUE.CNT>=2) then -- 2-byte instructions

        ---------------------------------------------------------------------------------------------------------------------------------
        --*****************************************************************************************************************************--
        --                                                      1-byte instructions                                                    --
        --*****************************************************************************************************************************--
        ---------------------------------------------------------------------------------------------------------------------------------
        if (IQUEUE.CNT>=1) then -- 1-byte instructions
    end if; -- if DBGMODE=0...
end if; -- if not stopped or halted
PC := PC + NUM_BYTES;                        -- update PC after instruction
IQUEUE.RDPOS := IQUEUE.RDPOS + NUM_BYTES;    -- update QUEUE read pointer
IQUEUE.CNT := IQUEUE.CNT - NUM_BYTES;        -- update QUEUE available bytes
if (OCD.SINGLESTEP='1') then                                                      -- if we are stepping instructions
    if (NUM_BYTES/=0 or IQUEUE.FETCH_STATE=F_ADDR) then OCD.SINGLESTEP:='0';    -- if a instruction was decoded, reset step flag
    end if;
end if;
```

FPz8 - Decoder



```
if (IQUEUE.CNT>=2) then -- 2-byte instructions
    if (IQUEUE.QUEUE(IQUEUE.RDPOS)(3 downto 0)=x"C") then   ------------------------------------------------- LD r,IMM instruction
        FRAB <= ADDRESSER4(IQUEUE.QUEUE(IQUEUE.RDPOS)(7 downto 4));
        DATAWRITE(ADDRESSER4(IQUEUE.QUEUE(IQUEUE.RDPOS)(7 downto 4)),IQUEUE.QUEUE(IQUEUE.RDPOS+1));
        NUM_BYTES := 2;
        CPU_STATE := CPU_STORE;
    elsif (IQUEUE.QUEUE(IQUEUE.RDPOS)(3 downto 0)=x"B") then    -------------------------------------------- JR cc,RelativeAddress
        PC := PC + 2;
        if (CONDITIONCODE(IQUEUE.QUEUE(IQUEUE.RDPOS)(7 downto 4))='1') then
            PC := ADDER16(PC,IQUEUE.QUEUE(IQUEUE.RDPOS+1));
            IQUEUE.FETCH_STATE := F_ADDR;
        else
            IQUEUE.RDPOS := IQUEUE.RDPOS + 2;
            IQUEUE.CNT := IQUEUE.CNT - 2;
        end if;
        CPU_STATE := CPU_DECOD;
    elsif (IQUEUE.QUEUE(IQUEUE.RDPOS)(3 downto 0)=x"A") then    ------------------------------------------- DJNZ r,RelativeAddress
        FRAB <= ADDRESSER4(IQUEUE.QUEUE(IQUEUE.RDPOS)(7 downto 4));
        PC := PC + 2;
        DEST_ADDR16 := ADDER16(PC,IQUEUE.QUEUE(IQUEUE.RDPOS+1));
        IQUEUE.RDPOS := IQUEUE.RDPOS + 2;
        IQUEUE.CNT := IQUEUE.CNT - 2;
        IQUEUE.FETCH_STATE := F_ADDR;
        CPU_STATE := CPU_DJNZ;
    elsif (IQUEUE.QUEUE(IQUEUE.RDPOS)(3 downto 0)=x"3") then    -------------------------------------------- column 3 instructions
        case IQUEUE.QUEUE(IQUEUE.RDPOS)(7 downto 4) is
            when x"8" =>    ------------------------------------------------------------------------ LDEI Ir1,Irr2 instruction
            when x"9" =>    ------------------------------------------------------------------------ LDEI Ir2,Irr1 instruction
            when x"C" =>    ------------------------------------------------------------------------ LDCI Ir1,Irr2 instruction
                RESULT(3 downto 0) := IQUEUE.QUEUE(IQUEUE.RDPOS+1)(3 downto 0);
                FRAB <= ADDRESSER4(IQUEUE.QUEUE(IQUEUE.RDPOS+1)(7 downto 4));
                NUM_BYTES := 2;
                CAN_FETCH := '0';
                LU_INSTRUCTION := '0';  -- indicates it is a read from program memory
                WORD_DATA := '1';       -- indicates it is a LDCI instruction
                CPU_STATE := CPU_LDPTOIM;
            when x"D" =>    ------------------------------------------------------------------------ LDCI Ir2,Irr1 instruction
                RESULT(3 downto 0) := IQUEUE.QUEUE(IQUEUE.RDPOS+1)(3 downto 0);
                FRAB <= ADDRESSER4(IQUEUE.QUEUE(IQUEUE.RDPOS+1)(7 downto 4));
                NUM_BYTES := 2;
                CAN_FETCH := '0';
                LU_INSTRUCTION := '1';  -- indicates it is a write onto program memory
                WORD_DATA := '1';       -- indicates it is a LDCI instruction
                CPU_STATE := CPU_LDPTOIM;
```

FPz8 - Decoder



```
if (IQUEUE.CNT>=3) then -- 3-byte instructions
    if (IQUEUE.QUEUE(IQUEUE.RDPOS)(3 downto 0)=x"D") then   -------------------------------------------------- JP cc,DirectAddress
        if (CONDITIONCODE(IQUEUE.QUEUE(IQUEUE.RDPOS)(7 downto 4))='1') then
            PC := IQUEUE.QUEUE(IQUEUE.RDPOS+1) & IQUEUE.QUEUE(IQUEUE.RDPOS+2);
            IQUEUE.FETCH_STATE := F_ADDR;
        else
            NUM_BYTES := 3;
        end if;
    elsif (IQUEUE.QUEUE(IQUEUE.RDPOS)(3 downto 0)=x"9") then    -------------------------------------------- column 9 instructions
        if (IQUEUE.QUEUE(IQUEUE.RDPOS)(7 downto 4)=x"8") then   ----------------------------------------- LDX rr1,r2,X instruction
            FRAB <= ADDRESSER4(IQUEUE.QUEUE(IQUEUE.RDPOS+1)(3 downto 0));                        -- source address
            DEST_ADDR := ADDRESSER4(IQUEUE.QUEUE(IQUEUE.RDPOS+1)(7 downto 4));                   -- dest address
            RESULT := IQUEUE.QUEUE(IQUEUE.RDPOS+2);                                              -- RESULT = offset (X)
            NUM_BYTES := 3;
            CPU_STATE := CPU_XRRD;
        elsif (IQUEUE.QUEUE(IQUEUE.RDPOS)(7 downto 4)=x"9") then    ------------------------------------ LEA rr1,rr2,X instruction
            FRAB <= ADDRESSER4(IQUEUE.QUEUE(IQUEUE.RDPOS+1)(3 downto 0));                        -- source address
            DEST_ADDR := ADDRESSER4(IQUEUE.QUEUE(IQUEUE.RDPOS+1)(7 downto 4));                   -- dest address
            RESULT := IQUEUE.QUEUE(IQUEUE.RDPOS+2);
            NUM_BYTES := 3;
            CPU_STATE := CPU_XRRTORR;
        end if;
    elsif (IQUEUE.QUEUE(IQUEUE.RDPOS)(3 downto 0)=x"8") then    -------------------------------------------- column 8 instructions
        if (IQUEUE.QUEUE(IQUEUE.RDPOS)(7 downto 4)=x"8") then   ----------------------------------------- LDX r1,rr2,X instruction
            FRAB <= ADDRESSER4(IQUEUE.QUEUE(IQUEUE.RDPOS+1)(3 downto 0));                        -- source address
            DEST_ADDR := ADDRESSER4(IQUEUE.QUEUE(IQUEUE.RDPOS+1)(7 downto 4));                   -- dest address
            RESULT := IQUEUE.QUEUE(IQUEUE.RDPOS+2);                                              -- RESULT = offset (X)
            NUM_BYTES := 3;
            CPU_STATE := CPU_XRRS;
        elsif (IQUEUE.QUEUE(IQUEUE.RDPOS)(7 downto 4)=x"9") then    -------------------------------------- LEA r1,r2,X instruction
            FRAB <= ADDRESSER4(IQUEUE.QUEUE(IQUEUE.RDPOS+1)(3 downto 0));                        -- source address
            DEST_ADDR := ADDRESSER4(IQUEUE.QUEUE(IQUEUE.RDPOS+1)(7 downto 4));                   -- dest address
            RESULT := IQUEUE.QUEUE(IQUEUE.RDPOS+2);
            NUM_BYTES := 3;
            CPU_STATE := CPU_XRTOM;
        elsif (IQUEUE.QUEUE(IQUEUE.RDPOS)(7 downto 4)=x"C") then    ---------------------------------------- PUSHX ER2 instruction
            SP := SP - 1;
            FRAB <= ADDRESSER12(IQUEUE.QUEUE(IQUEUE.RDPOS+1)&IQUEUE.QUEUE(IQUEUE.RDPOS+2)(7 downto 4));
            DEST_ADDR := SP;
            NUM_BYTES := 3;
            CPU_STATE := CPU_TMA;
```

FPz8 - Decoder

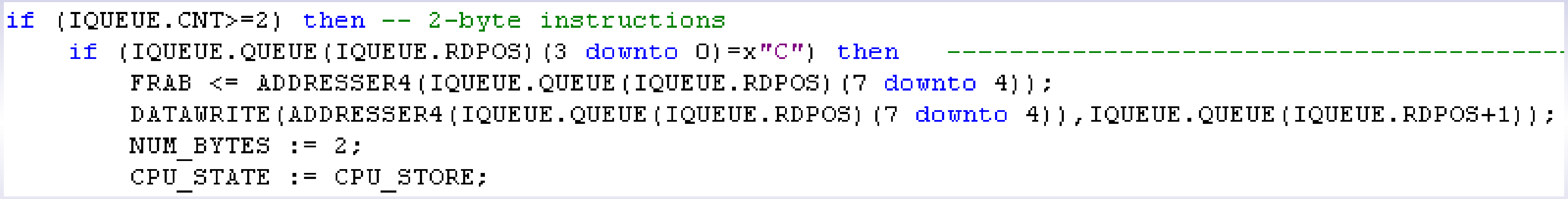
LD r1,IM

Carrega um valor constante de 8 bits no registrador r1

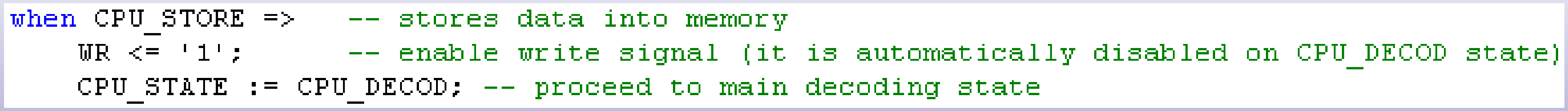
Tamanho: 2 bytes
Ciclos: 2

FPz8 - Decoder

LD r1,IM



```
if (IQUEUE.CNT>=2) then -- 2-byte instructions
    if (IQUEUE.QUEUE(IQUEUE.RDPOS)(3 downto 0)=x"C") then   ------------------------------------
        FRAB <= ADDRESSER4(IQUEUE.QUEUE(IQUEUE.RDPOS)(7 downto 4));
        DATAWRITE(ADDRESSER4(IQUEUE.QUEUE(IQUEUE.RDPOS)(7 downto 4)),IQUEUE.QUEUE(IQUEUE.RDPOS+1));
        NUM_BYTES := 2;
        CPU_STATE := CPU_STORE;
```



```
when CPU_STORE =>    -- stores data into memory
    WR <= '1';       -- enable write signal (it is automatically disabled on CPU_DECOD state)
    CPU_STATE := CPU_DECOD; -- proceed to main decoding state
```

FPz8 - Decoder

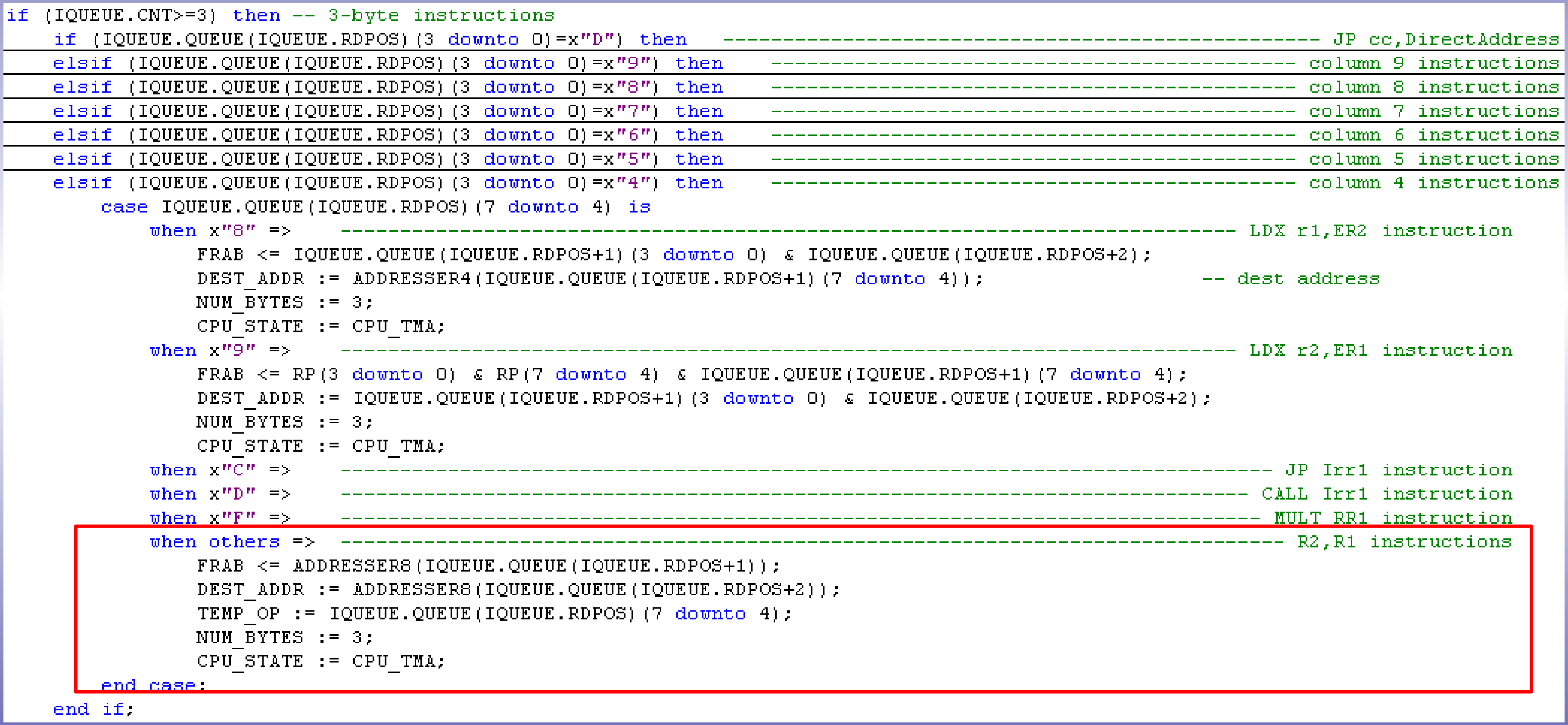
ADD R2,R1

Adiciona o conteúdo de R1 ao conteúdo de R2
(resultado em R2)

Tamanho: 3 bytes
Ciclos: 3

FPz8 - Decoder

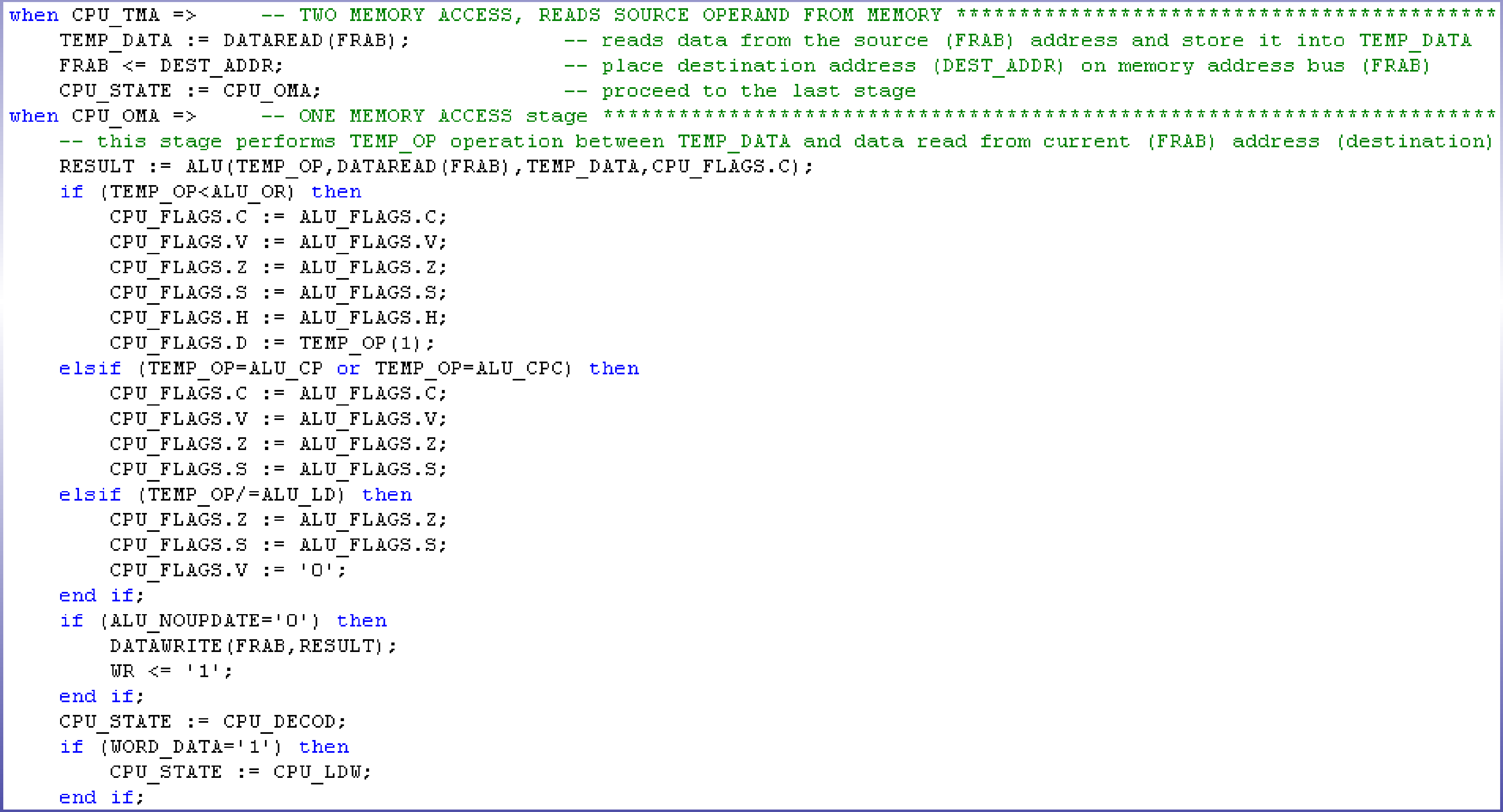
ADD R2,R1



```
if (IQUEUE.CNT>=3) then -- 3-byte instructions
    if (IQUEUE.QUEUE(IQUEUE.RDPOS)(3 downto 0)=x"D") then   ------------------------------------------------------------ JP cc,DirectAddress
    elsif (IQUEUE.QUEUE(IQUEUE.RDPOS)(3 downto 0)=x"9") then    --------------------------------------------------------- column 9 instructions
    elsif (IQUEUE.QUEUE(IQUEUE.RDPOS)(3 downto 0)=x"8") then    --------------------------------------------------------- column 8 instructions
    elsif (IQUEUE.QUEUE(IQUEUE.RDPOS)(3 downto 0)=x"7") then    --------------------------------------------------------- column 7 instructions
    elsif (IQUEUE.QUEUE(IQUEUE.RDPOS)(3 downto 0)=x"6") then    --------------------------------------------------------- column 6 instructions
    elsif (IQUEUE.QUEUE(IQUEUE.RDPOS)(3 downto 0)=x"5") then    --------------------------------------------------------- column 5 instructions
    elsif (IQUEUE.QUEUE(IQUEUE.RDPOS)(3 downto 0)=x"4") then    --------------------------------------------------------- column 4 instructions
        case IQUEUE.QUEUE(IQUEUE.RDPOS)(7 downto 4) is
            when x"8" =>    ----------------------------------------------------------------------------- LDX r1,ER2 instruction
                FRAB <= IQUEUE.QUEUE(IQUEUE.RDPOS+1)(3 downto 0) & IQUEUE.QUEUE(IQUEUE.RDPOS+2);
                DEST_ADDR := ADDRESSER4(IQUEUE.QUEUE(IQUEUE.RDPOS+1)(7 downto 4));          -- dest address
                NUM_BYTES := 3;
                CPU_STATE := CPU_TMA;
            when x"9" =>    ----------------------------------------------------------------------------- LDX r2,ER1 instruction
                FRAB <= RP(3 downto 0) & RP(7 downto 4) & IQUEUE.QUEUE(IQUEUE.RDPOS+1)(7 downto 4);
                DEST_ADDR := IQUEUE.QUEUE(IQUEUE.RDPOS+1)(3 downto 0) & IQUEUE.QUEUE(IQUEUE.RDPOS+2);
                NUM_BYTES := 3;
                CPU_STATE := CPU_TMA;
            when x"C" =>    -------------------------------------------------------------------------------- JP Irr1 instruction
            when x"D" =>    ------------------------------------------------------------------------------ CALL Irr1 instruction
            when x"F" =>    ------------------------------------------------------------------------------ MULT RR1 instruction
            when others =>  ------------------------------------------------------------------------------- R2,R1 instructions
                FRAB <= ADDRESSER8(IQUEUE.QUEUE(IQUEUE.RDPOS+1));
                DEST_ADDR := ADDRESSER8(IQUEUE.QUEUE(IQUEUE.RDPOS+2));
                TEMP_OP := IQUEUE.QUEUE(IQUEUE.RDPOS)(7 downto 4);
                NUM_BYTES := 3;
                CPU_STATE := CPU_TMA;
        end case;
    end if;
```

FPz8 - Decoder

ADD R2,R1

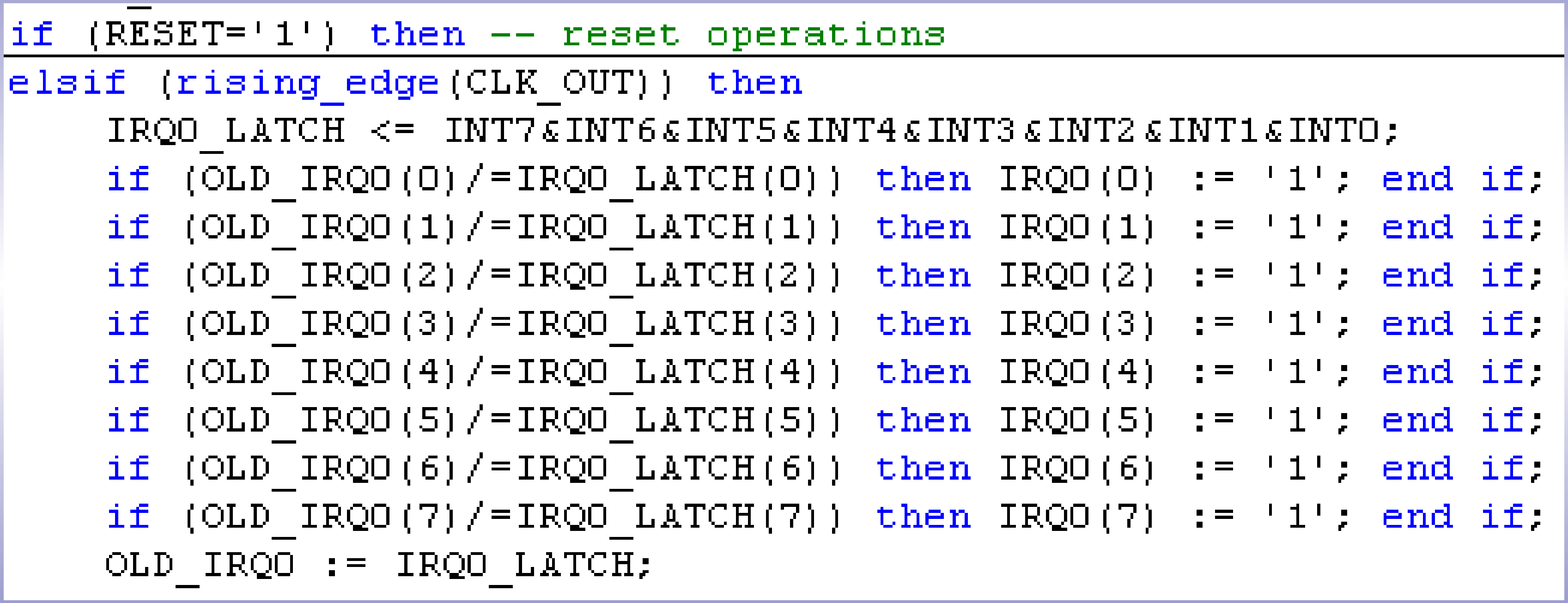


```
when CPU_TMA =>     -- TWO MEMORY ACCESS, READS SOURCE OPERAND FROM MEMORY *******************************************
    TEMP_DATA := DATAREAD(FRAB);            -- reads data from the source (FRAB) address and store it into TEMP_DATA
    FRAB <= DEST_ADDR;                      -- place destination address (DEST_ADDR) on memory address bus (FRAB)
    CPU_STATE := CPU_OMA;                   -- proceed to the last stage
when CPU_OMA =>     -- ONE MEMORY ACCESS stage ***********************************************************************
    -- this stage performs TEMP_OP operation between TEMP_DATA and data read from current (FRAB) address (destination)
    RESULT := ALU(TEMP_OP,DATAREAD(FRAB),TEMP_DATA,CPU_FLAGS.C);
    if (TEMP_OP<ALU_OR) then
        CPU_FLAGS.C := ALU_FLAGS.C;
        CPU_FLAGS.V := ALU_FLAGS.V;
        CPU_FLAGS.Z := ALU_FLAGS.Z;
        CPU_FLAGS.S := ALU_FLAGS.S;
        CPU_FLAGS.H := ALU_FLAGS.H;
        CPU_FLAGS.D := TEMP_OP(1);
    elsif (TEMP_OP=ALU_CP or TEMP_OP=ALU_CPC) then
        CPU_FLAGS.C := ALU_FLAGS.C;
        CPU_FLAGS.V := ALU_FLAGS.V;
        CPU_FLAGS.Z := ALU_FLAGS.Z;
        CPU_FLAGS.S := ALU_FLAGS.S;
    elsif (TEMP_OP/=ALU_LD) then
        CPU_FLAGS.Z := ALU_FLAGS.Z;
        CPU_FLAGS.S := ALU_FLAGS.S;
        CPU_FLAGS.V := '0';
    end if;
    if (ALU_NOUPDATE='0') then
        DATAWRITE(FRAB,RESULT);
        WR <= '1';
    end if;
    CPU_STATE := CPU_DECOD;
    if (WORD_DATA='1') then
        CPU_STATE := CPU_LDW;
    end if;
```

FPz8 – Interrupt Control

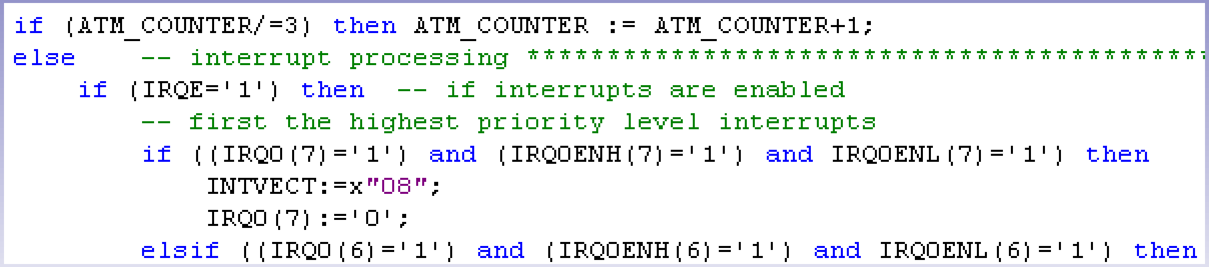
- Controlador de interrupções com três níveis de prioridade e múltiplos vetores
- Provoca a saída de um modo de baixo consumo

FPz8 – Interrupt Control

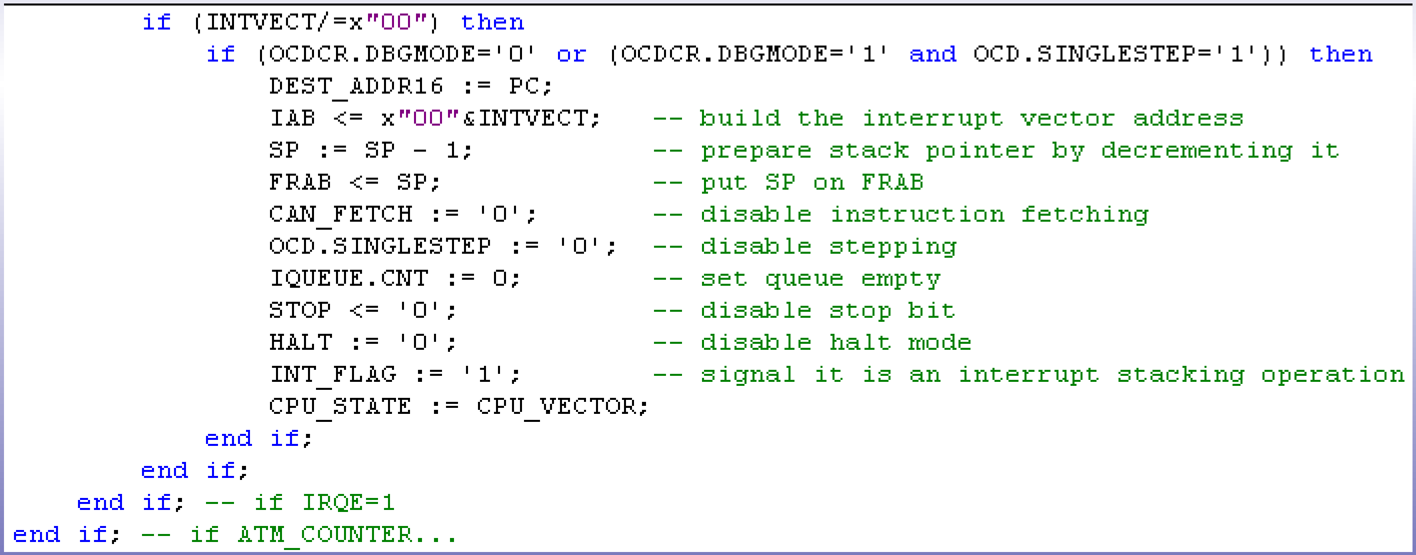


```
if (RESET='1') then -- reset operations
elsif (rising_edge(CLK_OUT)) then
    IRQ0_LATCH <= INT7&INT6&INT5&INT4&INT3&INT2&INT1&INT0;
    if (OLD_IRQ0(0)/=IRQ0_LATCH(0)) then IRQ0(0) := '1'; end if;
    if (OLD_IRQ0(1)/=IRQ0_LATCH(1)) then IRQ0(1) := '1'; end if;
    if (OLD_IRQ0(2)/=IRQ0_LATCH(2)) then IRQ0(2) := '1'; end if;
    if (OLD_IRQ0(3)/=IRQ0_LATCH(3)) then IRQ0(3) := '1'; end if;
    if (OLD_IRQ0(4)/=IRQ0_LATCH(4)) then IRQ0(4) := '1'; end if;
    if (OLD_IRQ0(5)/=IRQ0_LATCH(5)) then IRQ0(5) := '1'; end if;
    if (OLD_IRQ0(6)/=IRQ0_LATCH(6)) then IRQ0(6) := '1'; end if;
    if (OLD_IRQ0(7)/=IRQ0_LATCH(7)) then IRQ0(7) := '1'; end if;
    OLD_IRQ0 := IRQ0_LATCH;
```

FPz8 – Interrupt Control

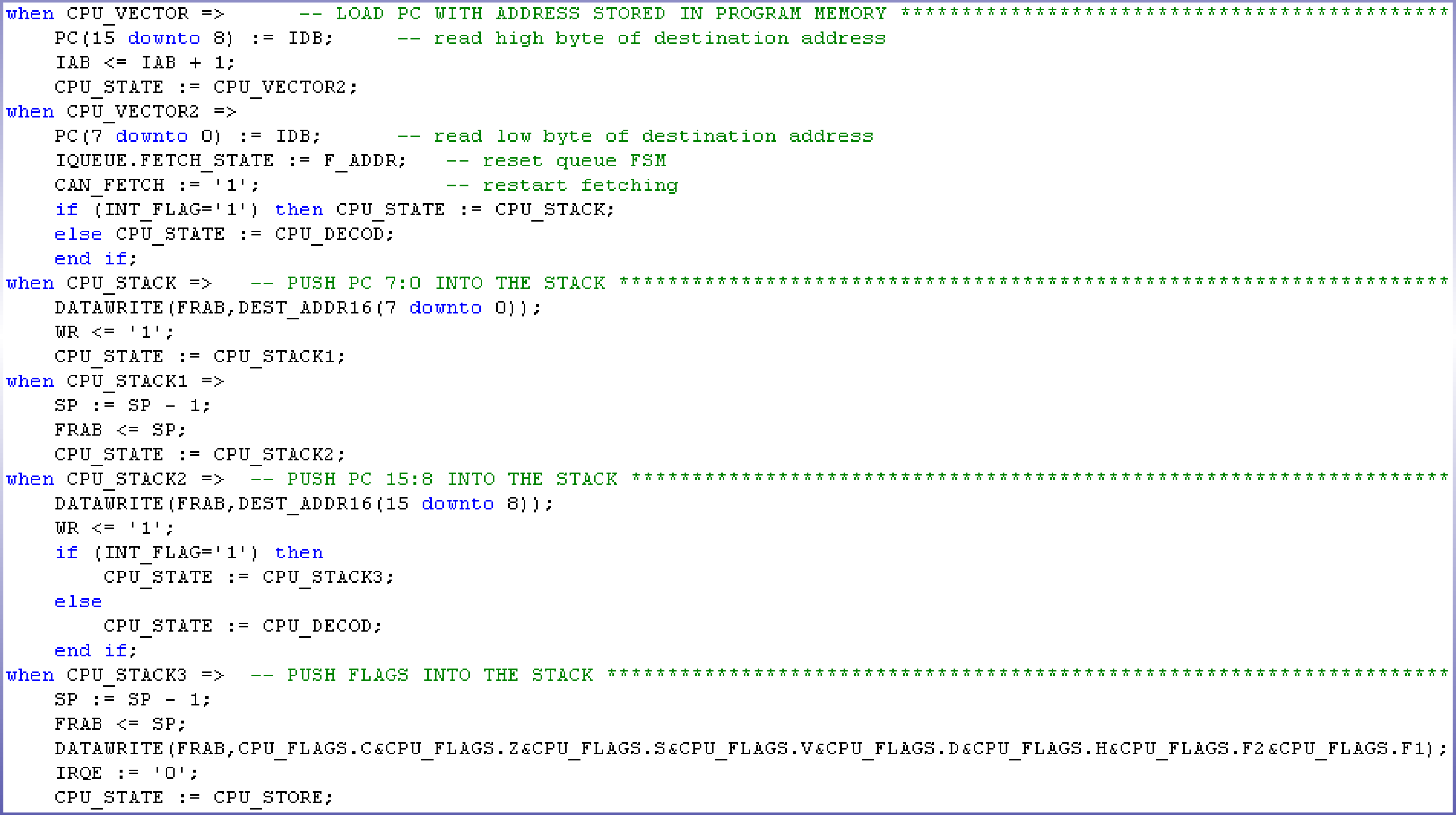


```
if (ATM_COUNTER/=3) then ATM_COUNTER := ATM_COUNTER+1;
else    -- interrupt processing *******************************************
    if (IRQE='1') then  -- if interrupts are enabled
        -- first the highest priority level interrupts
        if ((IRQ0(7)='1') and (IRQ0ENH(7)='1') and IRQ0ENL(7)='1') then
            INTVECT:=x"08";
            IRQ0(7):='0';
        elsif ((IRQ0(6)='1') and (IRQ0ENH(6)='1') and IRQ0ENL(6)='1') then
```



```
        if (INTVECT/=x"00") then
            if (OCDCR.DBGMODE='0' or (OCDCR.DBGMODE='1' and OCD.SINGLESTEP='1')) then
                DEST_ADDR16 := PC;
                IAB <= x"00"&INTVECT;   -- build the interrupt vector address
                SP := SP - 1;           -- prepare stack pointer by decrementing it
                FRAB <= SP;             -- put SP on FRAB
                CAN_FETCH := '0';       -- disable instruction fetching
                OCD.SINGLESTEP := '0';  -- disable stepping
                IQUEUE.CNT := 0;        -- set queue empty
                STOP <= '0';            -- disable stop bit
                HALT := '0';            -- disable halt mode
                INT_FLAG := '1';        -- signal it is an interrupt stacking operation
                CPU_STATE := CPU_VECTOR;
            end if;
        end if;
    end if; -- if IRQE=1
end if; -- if ATM_COUNTER...
```

FPz8 – Interrupt Control



```
when CPU_VECTOR =>      -- LOAD PC WITH ADDRESS STORED IN PROGRAM MEMORY *********************************************
    PC(15 downto 8) := IDB;     -- read high byte of destination address
    IAB <= IAB + 1;
    CPU_STATE := CPU_VECTOR2;
when CPU_VECTOR2 =>
    PC(7 downto 0) := IDB;      -- read low byte of destination address
    IQUEUE.FETCH_STATE := F_ADDR;   -- reset queue FSM
    CAN_FETCH := '1';               -- restart fetching
    if (INT_FLAG='1') then CPU_STATE := CPU_STACK;
    else CPU_STATE := CPU_DECOD;
    end if;
when CPU_STACK =>   -- PUSH PC 7:0 INTO THE STACK ********************************************************************
    DATAWRITE(FRAB,DEST_ADDR16(7 downto 0));
    WR <= '1';
    CPU_STATE := CPU_STACK1;
when CPU_STACK1 =>
    SP := SP - 1;
    FRAB <= SP;
    CPU_STATE := CPU_STACK2;
when CPU_STACK2 =>  -- PUSH PC 15:8 INTO THE STACK *******************************************************************
    DATAWRITE(FRAB,DEST_ADDR16(15 downto 8));
    WR <= '1';
    if (INT_FLAG='1') then
        CPU_STATE := CPU_STACK3;
    else
        CPU_STATE := CPU_DECOD;
    end if;
when CPU_STACK3 =>  -- PUSH FLAGS INTO THE STACK *********************************************************************
    SP := SP - 1;
    FRAB <= SP;
    DATAWRITE(FRAB,CPU_FLAGS.C&CPU_FLAGS.Z&CPU_FLAGS.S&CPU_FLAGS.V&CPU_FLAGS.D&CPU_FLAGS.H&CPU_FLAGS.F2&CPU_FLAGS.F1);
    IRQE := '0';
    CPU_STATE := CPU_STORE;
```

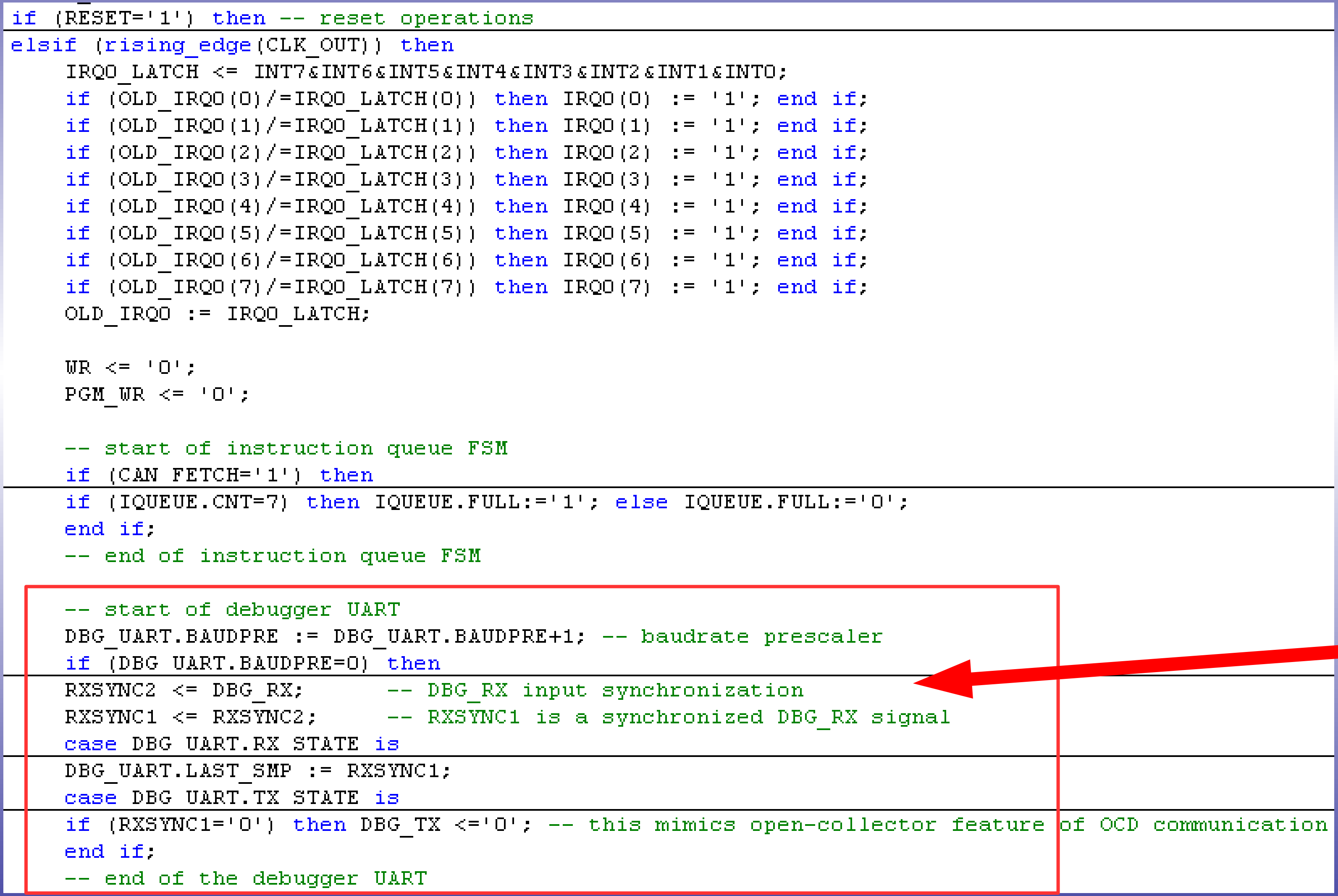
FPz8 – Interrupt Control

Número Vetor	Endereço	Pino de Interrupção
3	0x08	INT7
4	0x0A	INT6
5	0x0C	INT5
6	0x0E	INT4
7	0x10	INT3
8	0x12	INT2
9	0x14	INT1
10	0x16	INT0

FPz8 – On Chip Debugger

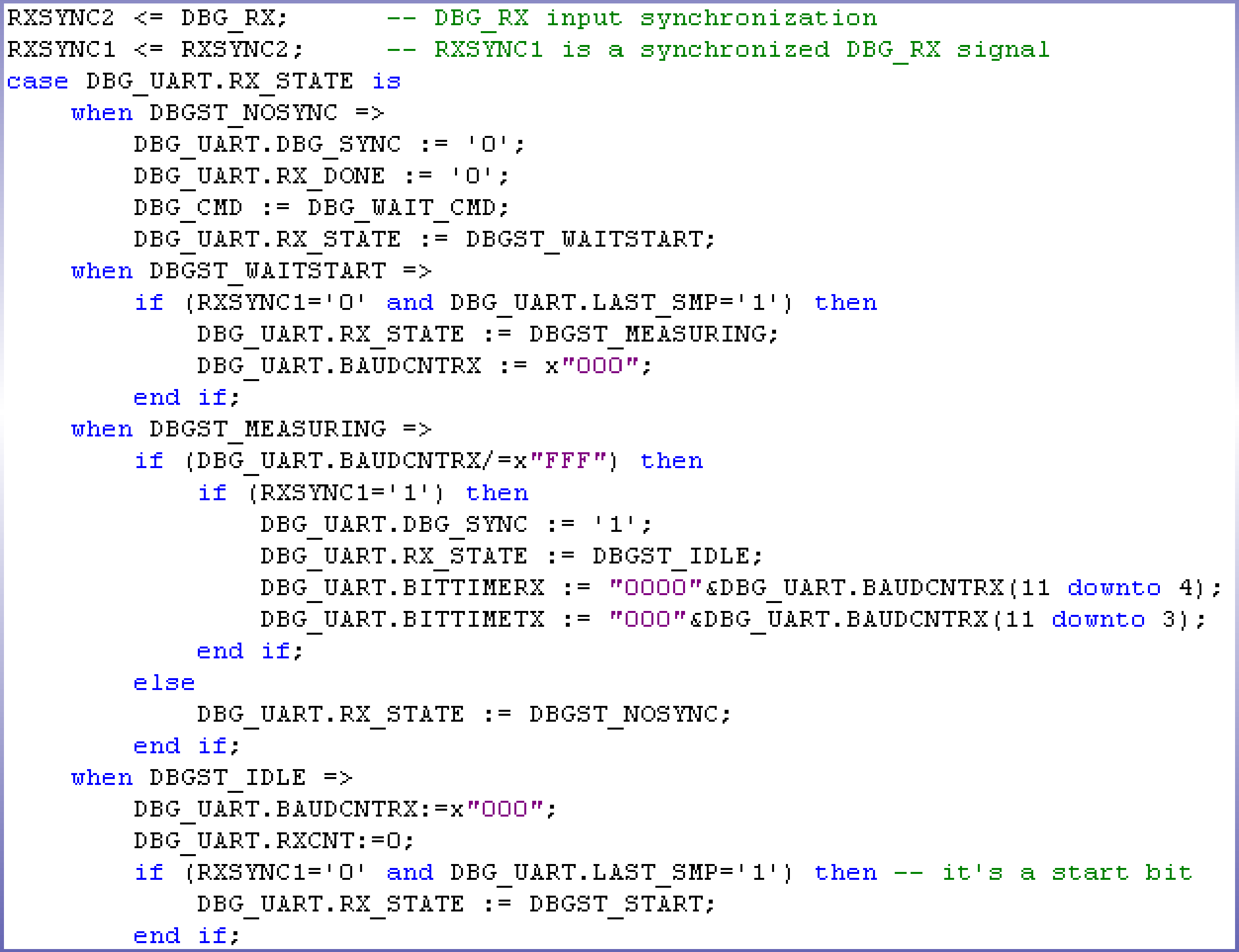
- Interpreta os comandos seriais recebidos do host
- Permite executar o programa, executar passo a passo, inserir instrução no fluxo, ler/escrever em qualquer posição da memória
- Nã estão implementados os comandos para leitura/escrita na memória de dados e nem o comando de cálculo de CRC da memória de programa

FPz8 – On Chip Debugger



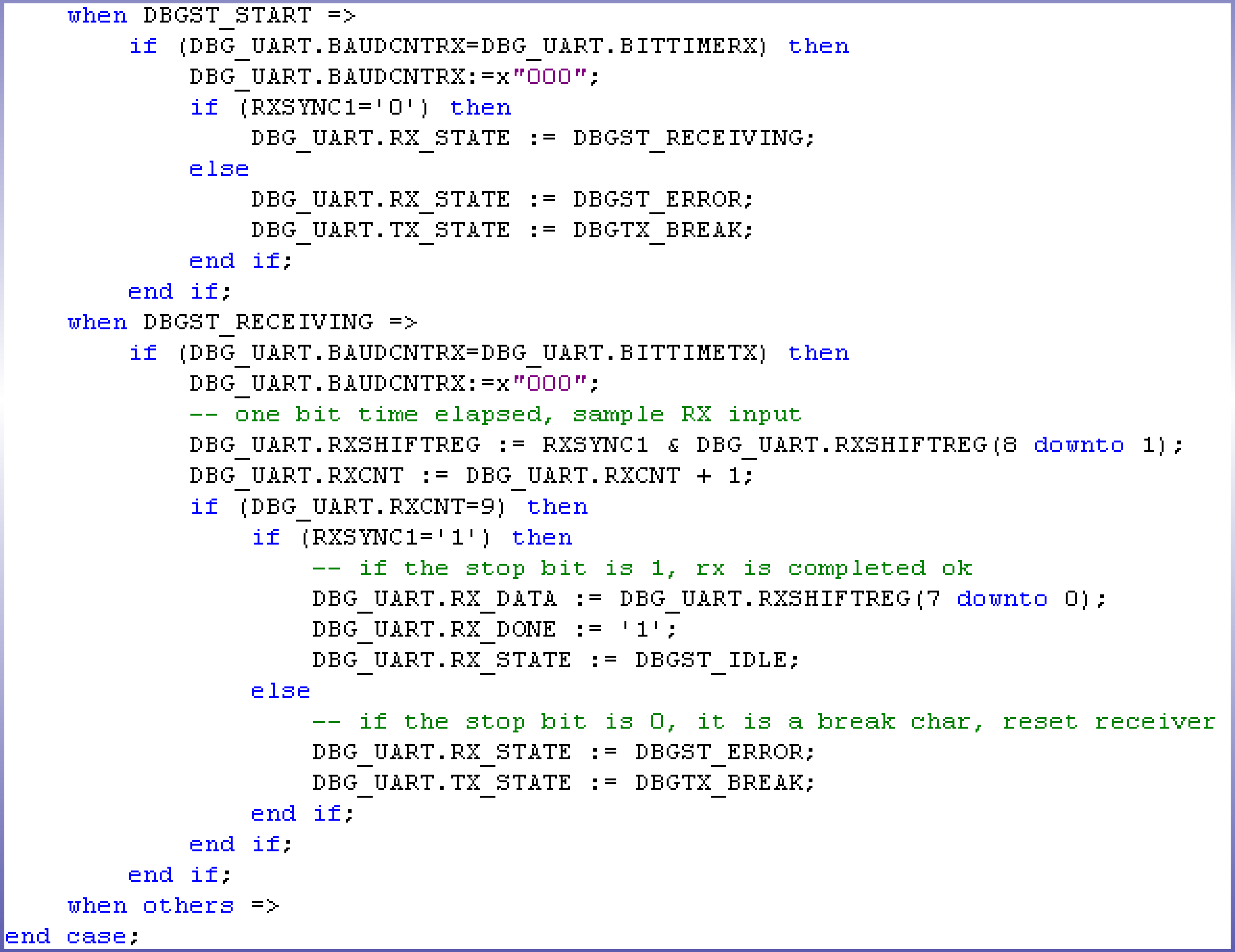
```
if (RESET='1') then -- reset operations
elsif (rising_edge(CLK_OUT)) then
    IRQ0_LATCH <= INT7&INT6&INT5&INT4&INT3&INT2&INT1&INT0;
    if (OLD_IRQ0(0)/=IRQ0_LATCH(0)) then IRQ0(0) := '1'; end if;
    if (OLD_IRQ0(1)/=IRQ0_LATCH(1)) then IRQ0(1) := '1'; end if;
    if (OLD_IRQ0(2)/=IRQ0_LATCH(2)) then IRQ0(2) := '1'; end if;
    if (OLD_IRQ0(3)/=IRQ0_LATCH(3)) then IRQ0(3) := '1'; end if;
    if (OLD_IRQ0(4)/=IRQ0_LATCH(4)) then IRQ0(4) := '1'; end if;
    if (OLD_IRQ0(5)/=IRQ0_LATCH(5)) then IRQ0(5) := '1'; end if;
    if (OLD_IRQ0(6)/=IRQ0_LATCH(6)) then IRQ0(6) := '1'; end if;
    if (OLD_IRQ0(7)/=IRQ0_LATCH(7)) then IRQ0(7) := '1'; end if;
    OLD_IRQ0 := IRQ0_LATCH;

    WR <= '0';
    PGM_WR <= '0';

    -- start of instruction queue FSM
    if (CAN_FETCH='1') then
    if (IQUEUE.CNT=7) then IQUEUE.FULL:='1'; else IQUEUE.FULL:='0';
    end if;
    -- end of instruction queue FSM

    -- start of debugger UART
    DBG_UART.BAUDPRE := DBG_UART.BAUDPRE+1; -- baudrate prescaler
    if (DBG_UART.BAUDPRE=0) then
    RXSYNC2 <= DBG_RX;        -- DBG_RX input synchronization
    RXSYNC1 <= RXSYNC2;       -- RXSYNC1 is a synchronized DBG_RX signal
    case DBG_UART.RX_STATE is
    DBG_UART.LAST_SMP := RXSYNC1;
    case DBG_UART.TX_STATE is
    if (RXSYNC1='0') then DBG_TX <='0'; -- this mimics open-collector feature of OCD communication
    end if;
    -- end of the debugger UART
```

FPz8 – On Chip Debugger



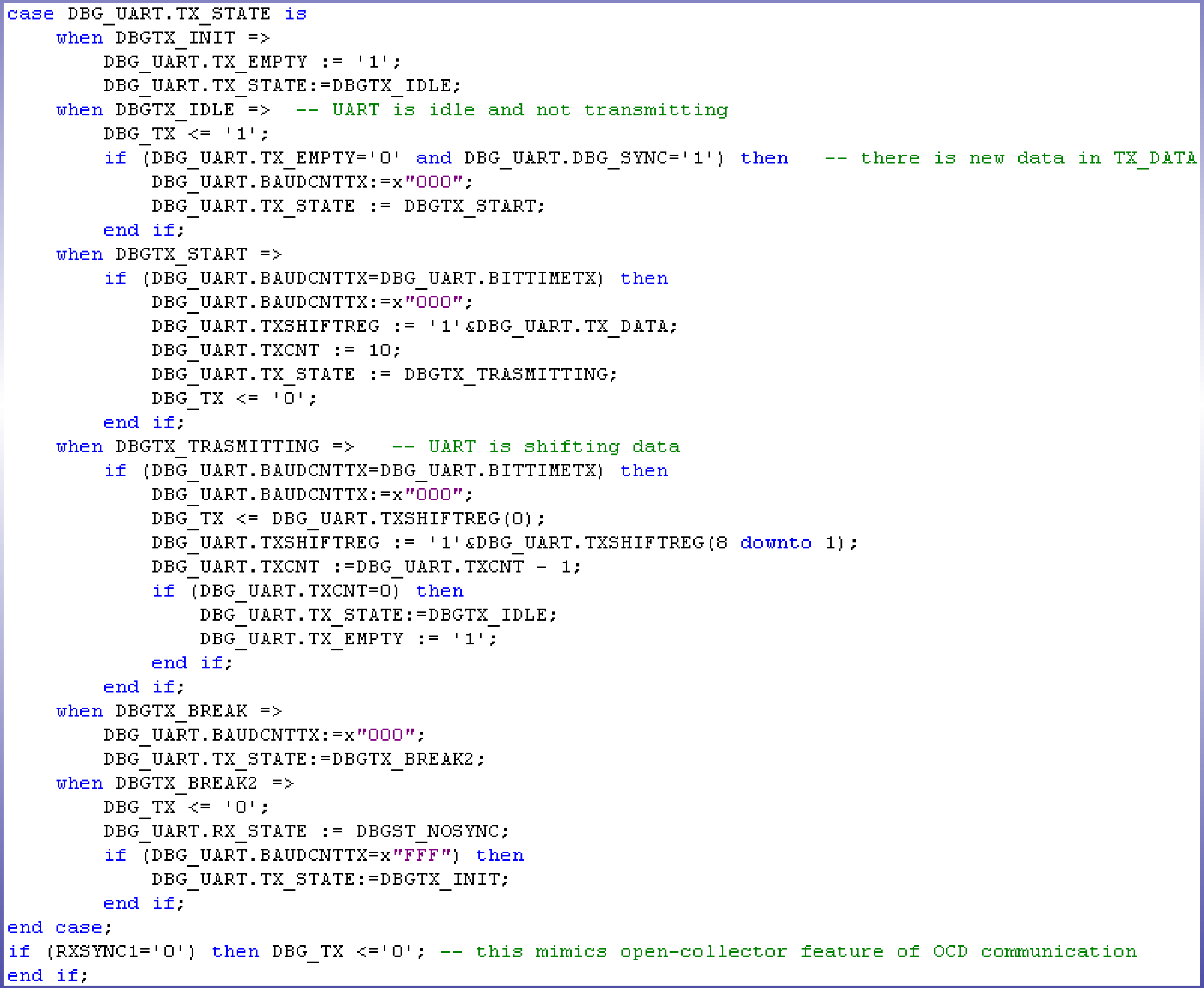
```
RXSYNC2 <= DBG_RX;        -- DBG_RX input synchronization
RXSYNC1 <= RXSYNC2;       -- RXSYNC1 is a synchronized DBG_RX signal
case DBG_UART.RX_STATE is
    when DBGST_NOSYNC =>
        DBG_UART.DBG_SYNC := '0';
        DBG_UART.RX_DONE := '0';
        DBG_CMD := DBG_WAIT_CMD;
        DBG_UART.RX_STATE := DBGST_WAITSTART;
    when DBGST_WAITSTART =>
        if (RXSYNC1='0' and DBG_UART.LAST_SMP='1') then
            DBG_UART.RX_STATE := DBGST_MEASURING;
            DBG_UART.BAUDCNTRX := x"000";
        end if;
    when DBGST_MEASURING =>
        if (DBG_UART.BAUDCNTRX/=x"FFF") then
            if (RXSYNC1='1') then
                DBG_UART.DBG_SYNC := '1';
                DBG_UART.RX_STATE := DBGST_IDLE;
                DBG_UART.BITTIMERX := "0000"&DBG_UART.BAUDCNTRX(11 downto 4);
                DBG_UART.BITTIMETX := "000"&DBG_UART.BAUDCNTRX(11 downto 3);
            end if;
        else
            DBG_UART.RX_STATE := DBGST_NOSYNC;
        end if;
    when DBGST_IDLE =>
        DBG_UART.BAUDCNTRX:=x"000";
        DBG_UART.RXCNT:=0;
        if (RXSYNC1='0' and DBG_UART.LAST_SMP='1') then -- it's a start bit
            DBG_UART.RX_STATE := DBGST_START;
        end if;
```

FPz8 – On Chip Debugger



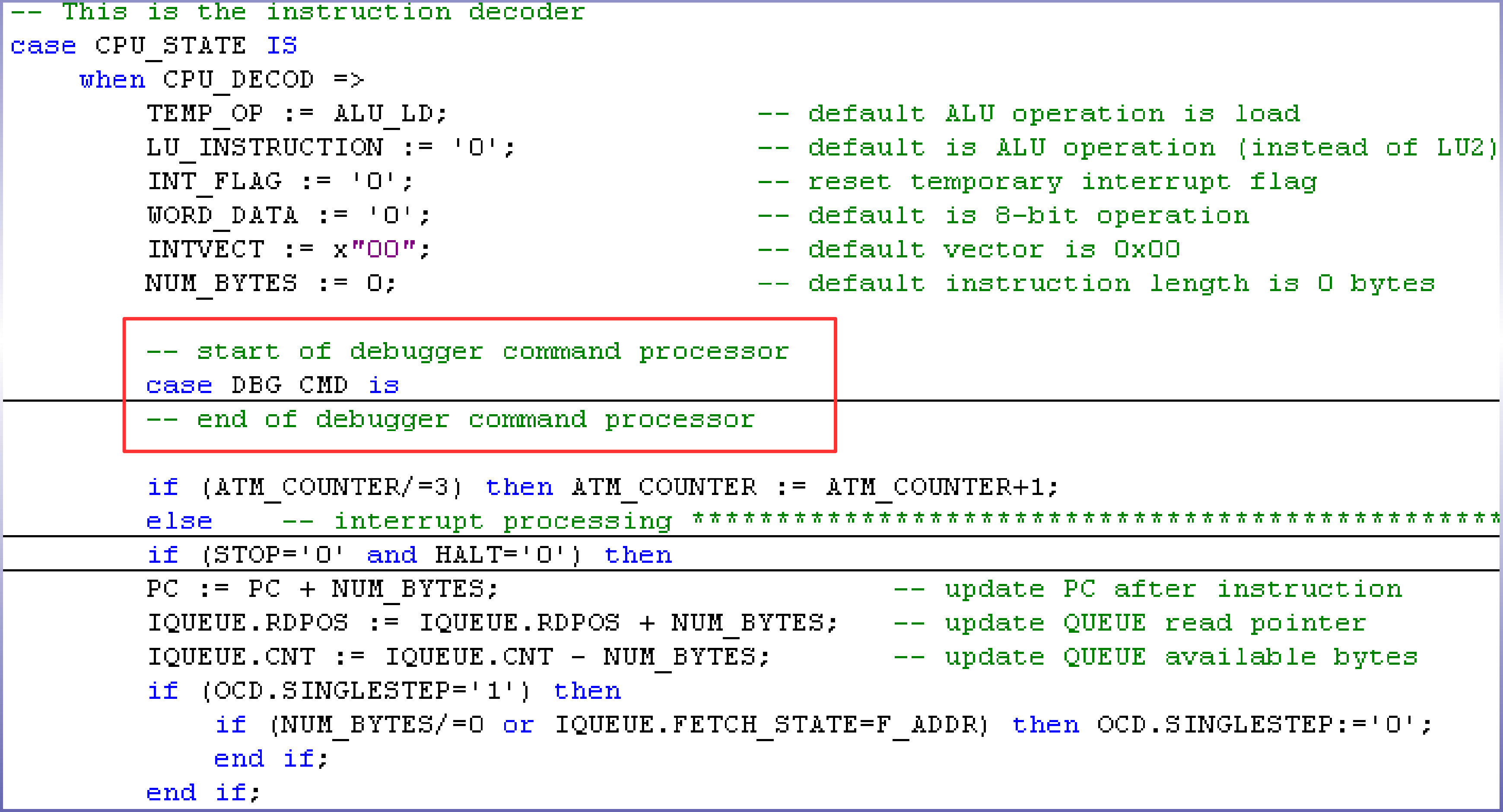
```vhdl
    when DBGST_START =>
        if (DBG_UART.BAUDCNTRX=DBG_UART.BITTIMERX) then
            DBG_UART.BAUDCNTRX:=x"000";
            if (RXSYNC1='0') then
                DBG_UART.RX_STATE := DBGST_RECEIVING;
            else
                DBG_UART.RX_STATE := DBGST_ERROR;
                DBG_UART.TX_STATE := DBGTX_BREAK;
            end if;
        end if;
    when DBGST_RECEIVING =>
        if (DBG_UART.BAUDCNTRX=DBG_UART.BITTIMETX) then
            DBG_UART.BAUDCNTRX:=x"000";
            -- one bit time elapsed, sample RX input
            DBG_UART.RXSHIFTREG := RXSYNC1 & DBG_UART.RXSHIFTREG(8 downto 1);
            DBG_UART.RXCNT := DBG_UART.RXCNT + 1;
            if (DBG_UART.RXCNT=9) then
                if (RXSYNC1='1') then
                    -- if the stop bit is 1, rx is completed ok
                    DBG_UART.RX_DATA := DBG_UART.RXSHIFTREG(7 downto 0);
                    DBG_UART.RX_DONE := '1';
                    DBG_UART.RX_STATE := DBGST_IDLE;
                else
                    -- if the stop bit is 0, it is a break char, reset receiver
                    DBG_UART.RX_STATE := DBGST_ERROR;
                    DBG_UART.TX_STATE := DBGTX_BREAK;
                end if;
            end if;
        end if;
    when others =>
end case;
```

FPz8 – On Chip Debugger



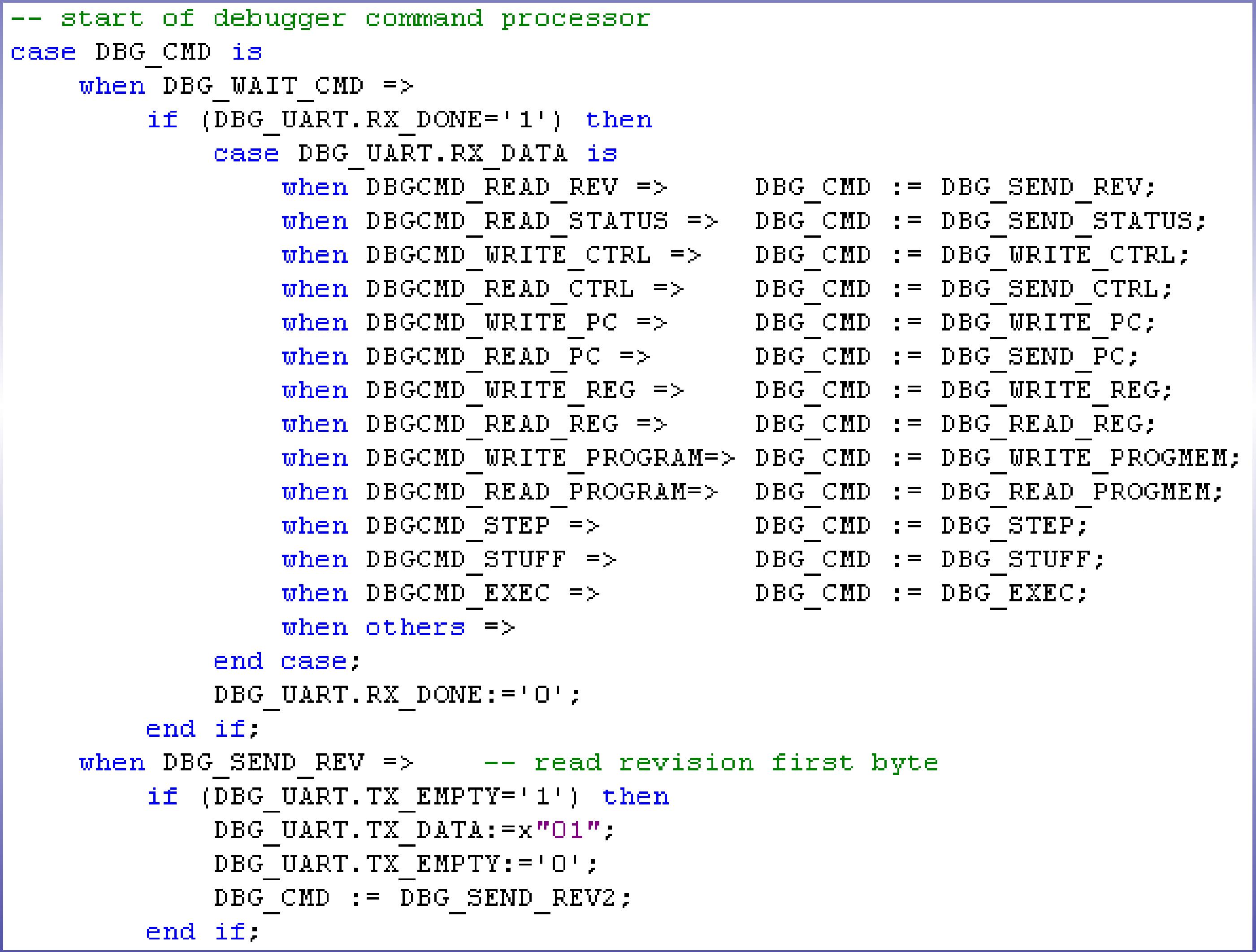
```
case DBG_UART.TX_STATE is
    when DBGTX_INIT =>
        DBG_UART.TX_EMPTY := '1';
        DBG_UART.TX_STATE:=DBGTX_IDLE;
    when DBGTX_IDLE =>  -- UART is idle and not transmitting
        DBG_TX <= '1';
        if (DBG_UART.TX_EMPTY='0' and DBG_UART.DBG_SYNC='1') then   -- there is new data in TX_DATA
            DBG_UART.BAUDCNTTX:=x"000";
            DBG_UART.TX_STATE := DBGTX_START;
        end if;
    when DBGTX_START =>
        if (DBG_UART.BAUDCNTTX=DBG_UART.BITTIMETX) then
            DBG_UART.BAUDCNTTX:=x"000";
            DBG_UART.TXSHIFTREG := '1'&DBG_UART.TX_DATA;
            DBG_UART.TXCNT := 10;
            DBG_UART.TX_STATE := DBGTX_TRASMITTING;
            DBG_TX <= '0';
        end if;
    when DBGTX_TRASMITTING =>   -- UART is shifting data
        if (DBG_UART.BAUDCNTTX=DBG_UART.BITTIMETX) then
            DBG_UART.BAUDCNTTX:=x"000";
            DBG_TX <= DBG_UART.TXSHIFTREG(0);
            DBG_UART.TXSHIFTREG := '1'&DBG_UART.TXSHIFTREG(8 downto 1);
            DBG_UART.TXCNT :=DBG_UART.TXCNT - 1;
            if (DBG_UART.TXCNT=0) then
                DBG_UART.TX_STATE:=DBGTX_IDLE;
                DBG_UART.TX_EMPTY := '1';
            end if;
        end if;
    when DBGTX_BREAK =>
        DBG_UART.BAUDCNTTX:=x"000";
        DBG_UART.TX_STATE:=DBGTX_BREAK2;
    when DBGTX_BREAK2 =>
        DBG_TX <= '0';
        DBG_UART.RX_STATE := DBGST_NOSYNC;
        if (DBG_UART.BAUDCNTTX=x"FFF") then
            DBG_UART.TX_STATE:=DBGTX_INIT;
        end if;
end case;
if (RXSYNC1='0') then DBG_TX <='0'; -- this mimics open-collector feature of OCD communication
end if;
```

FPz8 – On Chip Debugger



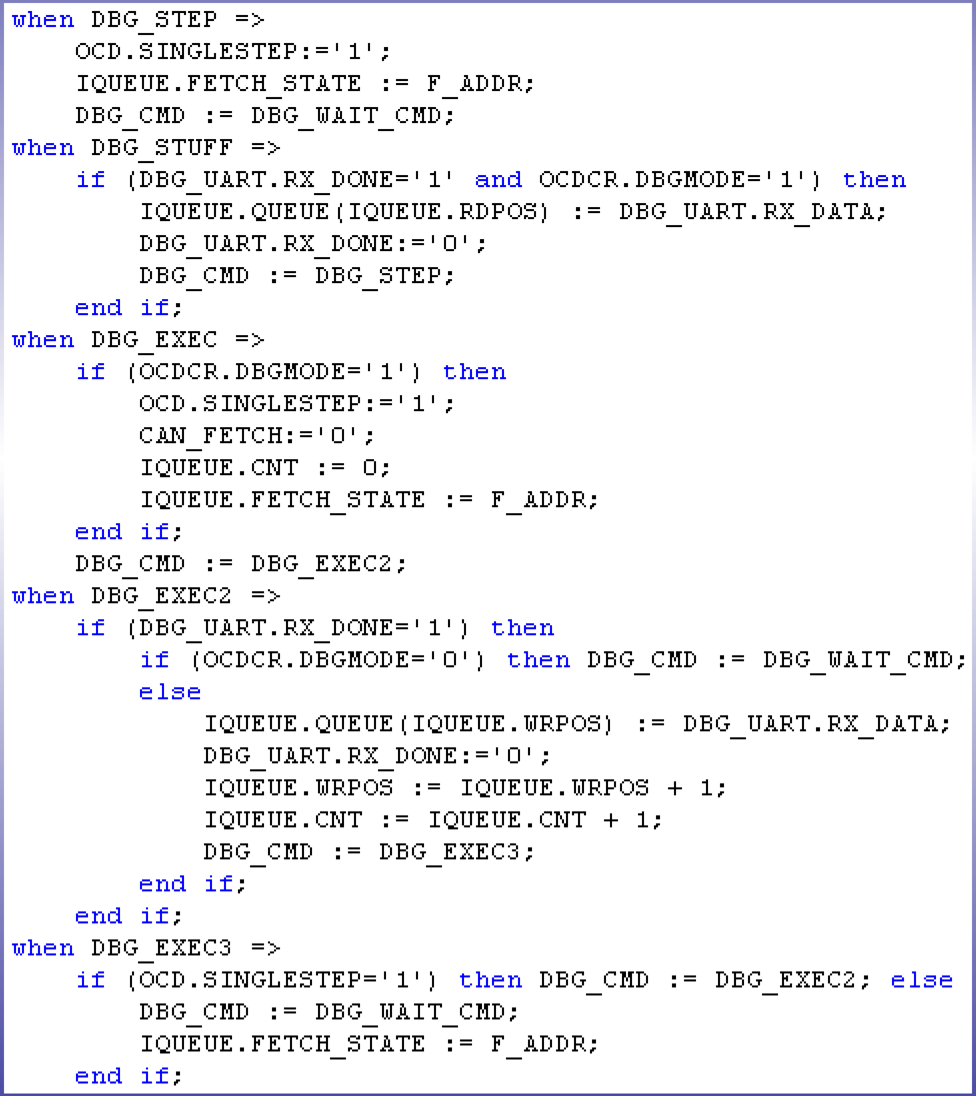
```
-- This is the instruction decoder
case CPU_STATE IS
    when CPU_DECOD =>
        TEMP_OP := ALU_LD;                  -- default ALU operation is load
        LU_INSTRUCTION := '0';              -- default is ALU operation (instead of LU2)
        INT_FLAG := '0';                    -- reset temporary interrupt flag
        WORD_DATA := '0';                   -- default is 8-bit operation
        INTVECT := x"00";                   -- default vector is 0x00
        NUM_BYTES := 0;                     -- default instruction length is 0 bytes

        -- start of debugger command processor
        case DBG_CMD is
        -- end of debugger command processor

        if (ATM_COUNTER/=3) then ATM_COUNTER := ATM_COUNTER+1;
        else    -- interrupt processing ****************************************
        if (STOP='0' and HALT='0') then
        PC := PC + NUM_BYTES;                       -- update PC after instruction
        IQUEUE.RDPOS := IQUEUE.RDPOS + NUM_BYTES;   -- update QUEUE read pointer
        IQUEUE.CNT := IQUEUE.CNT - NUM_BYTES;       -- update QUEUE available bytes
        if (OCD.SINGLESTEP='1') then
            if (NUM_BYTES/=0 or IQUEUE.FETCH_STATE=F_ADDR) then OCD.SINGLESTEP:='0';
            end if;
        end if;
```

FPz8 – On Chip Debugger



```
-- start of debugger command processor
case DBG_CMD is
    when DBG_WAIT_CMD =>
        if (DBG_UART.RX_DONE='1') then
            case DBG_UART.RX_DATA is
                when DBGCMD_READ_REV =>     DBG_CMD := DBG_SEND_REV;
                when DBGCMD_READ_STATUS =>  DBG_CMD := DBG_SEND_STATUS;
                when DBGCMD_WRITE_CTRL =>   DBG_CMD := DBG_WRITE_CTRL;
                when DBGCMD_READ_CTRL =>    DBG_CMD := DBG_SEND_CTRL;
                when DBGCMD_WRITE_PC =>     DBG_CMD := DBG_WRITE_PC;
                when DBGCMD_READ_PC =>      DBG_CMD := DBG_SEND_PC;
                when DBGCMD_WRITE_REG =>    DBG_CMD := DBG_WRITE_REG;
                when DBGCMD_READ_REG =>     DBG_CMD := DBG_READ_REG;
                when DBGCMD_WRITE_PROGRAM=> DBG_CMD := DBG_WRITE_PROGMEM;
                when DBGCMD_READ_PROGRAM=>  DBG_CMD := DBG_READ_PROGMEM;
                when DBGCMD_STEP =>         DBG_CMD := DBG_STEP;
                when DBGCMD_STUFF =>        DBG_CMD := DBG_STUFF;
                when DBGCMD_EXEC =>         DBG_CMD := DBG_EXEC;
                when others =>
            end case;
            DBG_UART.RX_DONE:='0';
        end if;
    when DBG_SEND_REV =>   -- read revision first byte
        if (DBG_UART.TX_EMPTY='1') then
            DBG_UART.TX_DATA:=x"01";
            DBG_UART.TX_EMPTY:='0';
            DBG_CMD := DBG_SEND_REV2;
        end if;
```

FPz8 – On Chip Debugger

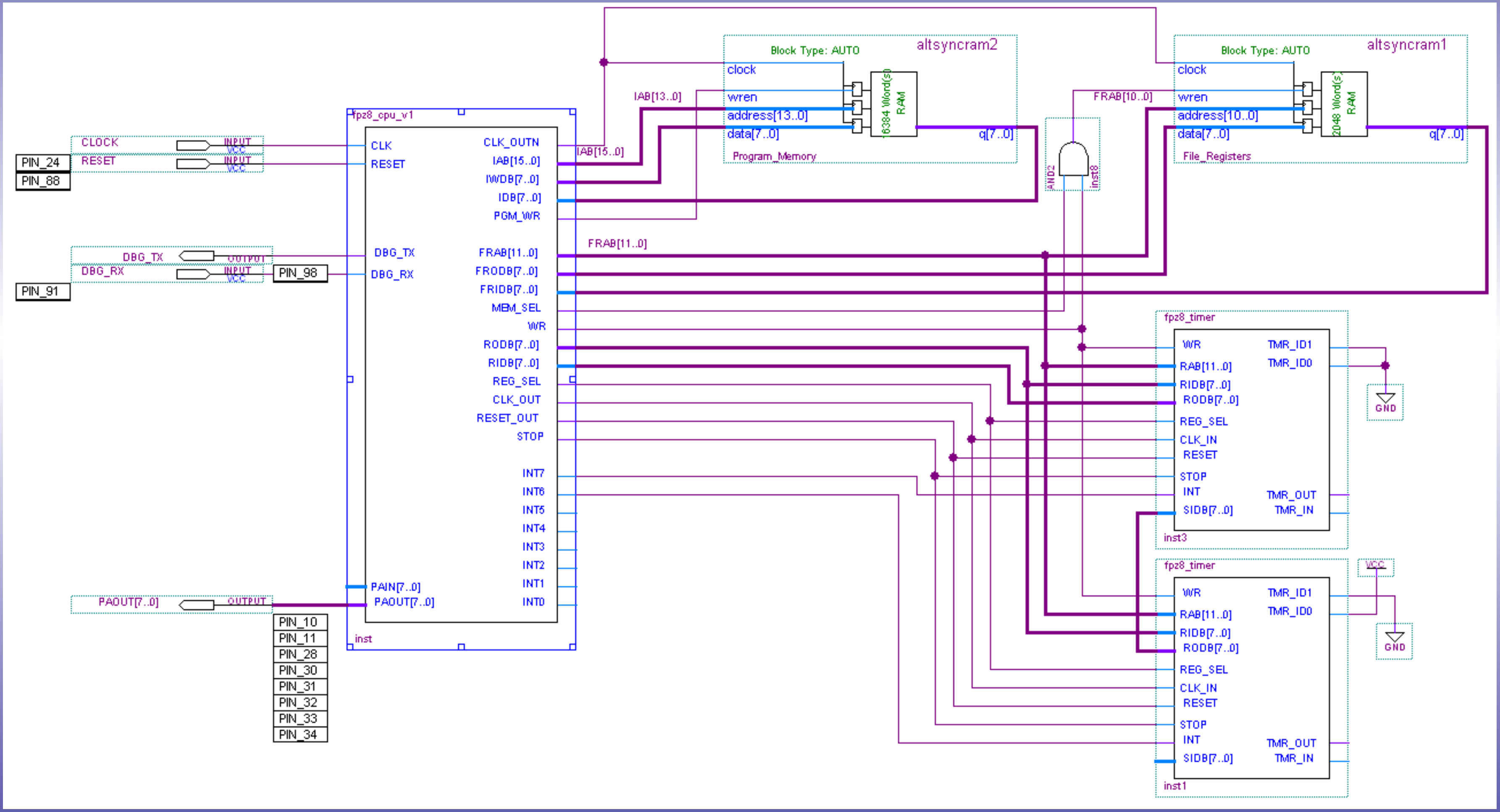


```
when DBG_STEP =>
    OCD.SINGLESTEP:='1';
    IQUEUE.FETCH_STATE := F_ADDR;
    DBG_CMD := DBG_WAIT_CMD;
when DBG_STUFF =>
    if (DBG_UART.RX_DONE='1' and OCDCR.DBGMODE='1') then
        IQUEUE.QUEUE(IQUEUE.RDPOS) := DBG_UART.RX_DATA;
        DBG_UART.RX_DONE:='0';
        DBG_CMD := DBG_STEP;
    end if;
when DBG_EXEC =>
    if (OCDCR.DBGMODE='1') then
        OCD.SINGLESTEP:='1';
        CAN_FETCH:='0';
        IQUEUE.CNT := 0;
        IQUEUE.FETCH_STATE := F_ADDR;
    end if;
    DBG_CMD := DBG_EXEC2;
when DBG_EXEC2 =>
    if (DBG_UART.RX_DONE='1') then
        if (OCDCR.DBGMODE='0') then DBG_CMD := DBG_WAIT_CMD;
        else
            IQUEUE.QUEUE(IQUEUE.WRPOS) := DBG_UART.RX_DATA;
            DBG_UART.RX_DONE:='0';
            IQUEUE.WRPOS := IQUEUE.WRPOS + 1;
            IQUEUE.CNT := IQUEUE.CNT + 1;
            DBG_CMD := DBG_EXEC3;
        end if;
    end if;
when DBG_EXEC3 =>
    if (OCD.SINGLESTEP='1') then DBG_CMD := DBG_EXEC2; else
        DBG_CMD := DBG_WAIT_CMD;
        IQUEUE.FETCH_STATE := F_ADDR;
    end if;
```

FPz8 – Estatísticas

- Velocidade máxima: aproximadamente 23.1MHz
- Ocupação do FPGA:
  - CPU completa: aproximadamente 4889 LEs, 522 registradores
  - CPU sem debugger: aprox. 4009 LEs, 365 regs
  - Timer básico (16 bits): aproximadamente 120 LEs, 61 registradores

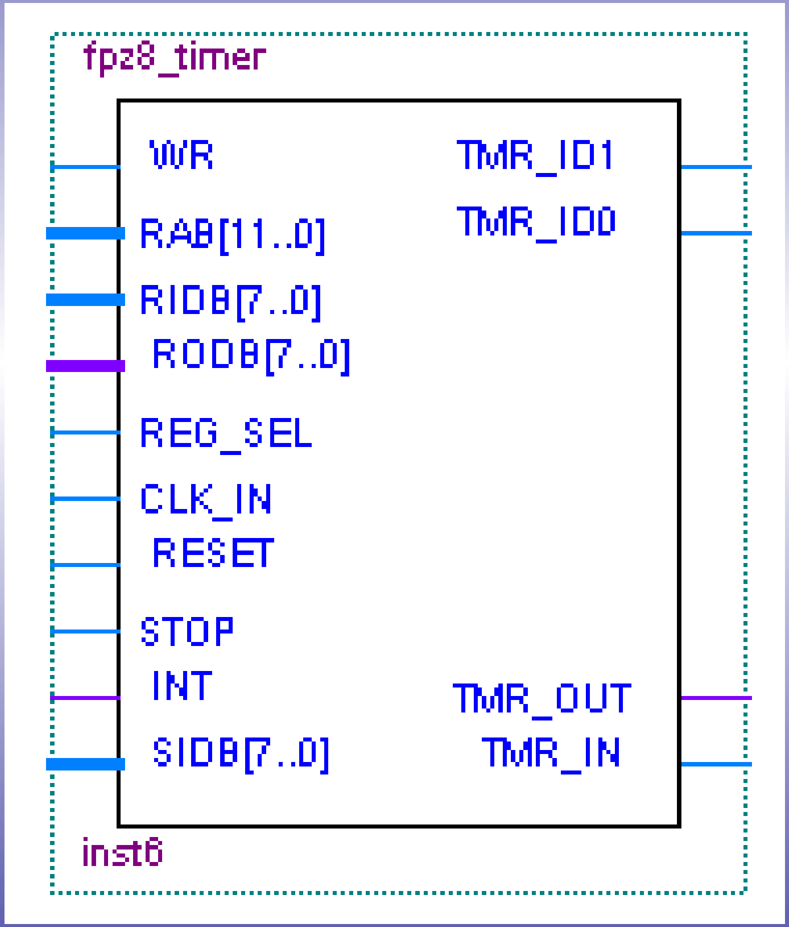
FPz8 – Exemplo



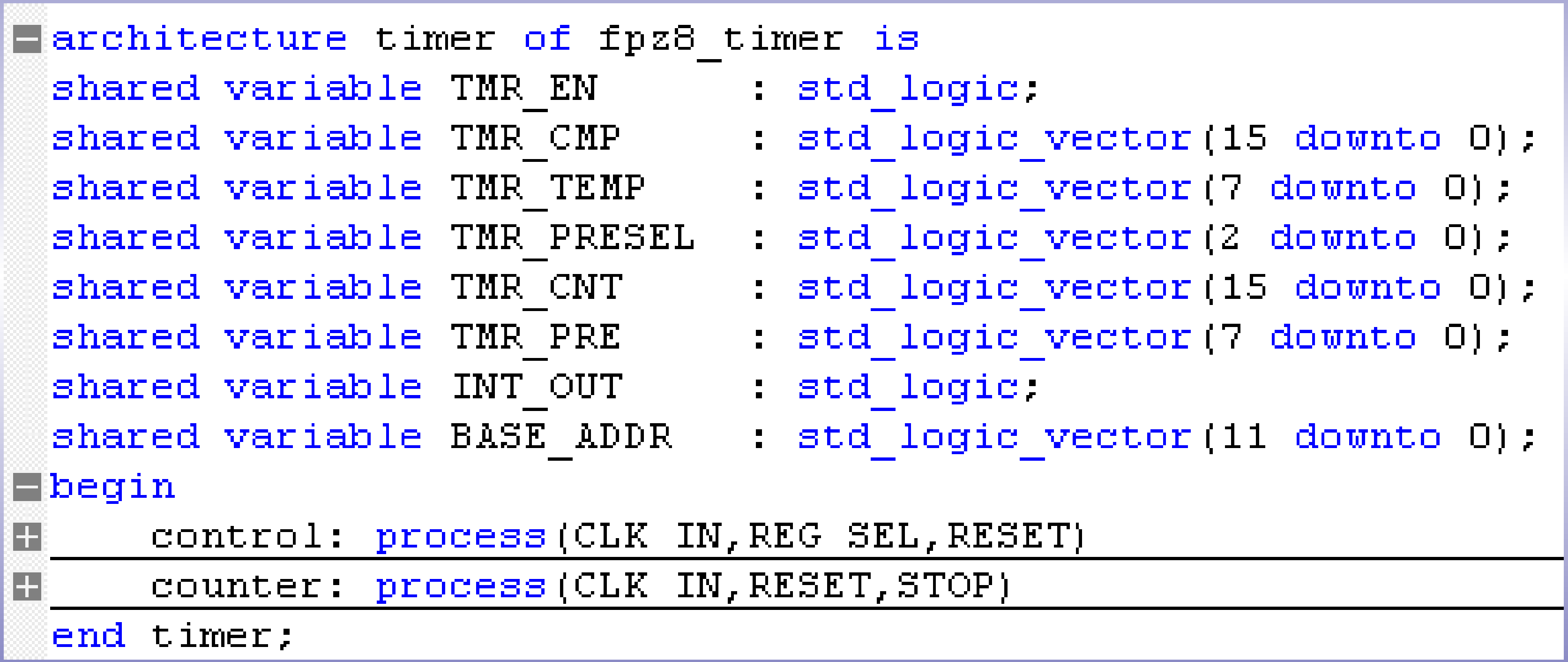
FPz8 – Exemplo

```
// Test application - blink leds on PAOUT
#include <ez8f1622.h>

void interrupt isr_timer0(){
        PAOUT ^= 1;             // toggle LED at PAOUT_0
}

void interrupt isr_timer1(){
        PAOUT ^= 2;             // toggle LED at PAOUT_1
}

void main(){
        PAOUT = 0;
        SET_VECTOR(3,isr_timer0);       // set interrupt vector for timer0 ISR (vector 3)
        SET_VECTOR(4,isr_timer1);       // set interrupt vector for timer1 ISR (vector 4)
        T0R=0x7FFF;                     // Timer0 Reload
        T0CTL1=0xB8;                    // Timer0 Control Register 0
        T1R = 0x7F00;                   // Timer1 Reload
        T1CTL1=0xB8;                    // Timer1 Control Register 0
        IRQ0ENL = 0xC0;                 // enable IRQs
        EI();                           // enable interrupts
        while(1);
}
```

FPz8 – Timer Básico

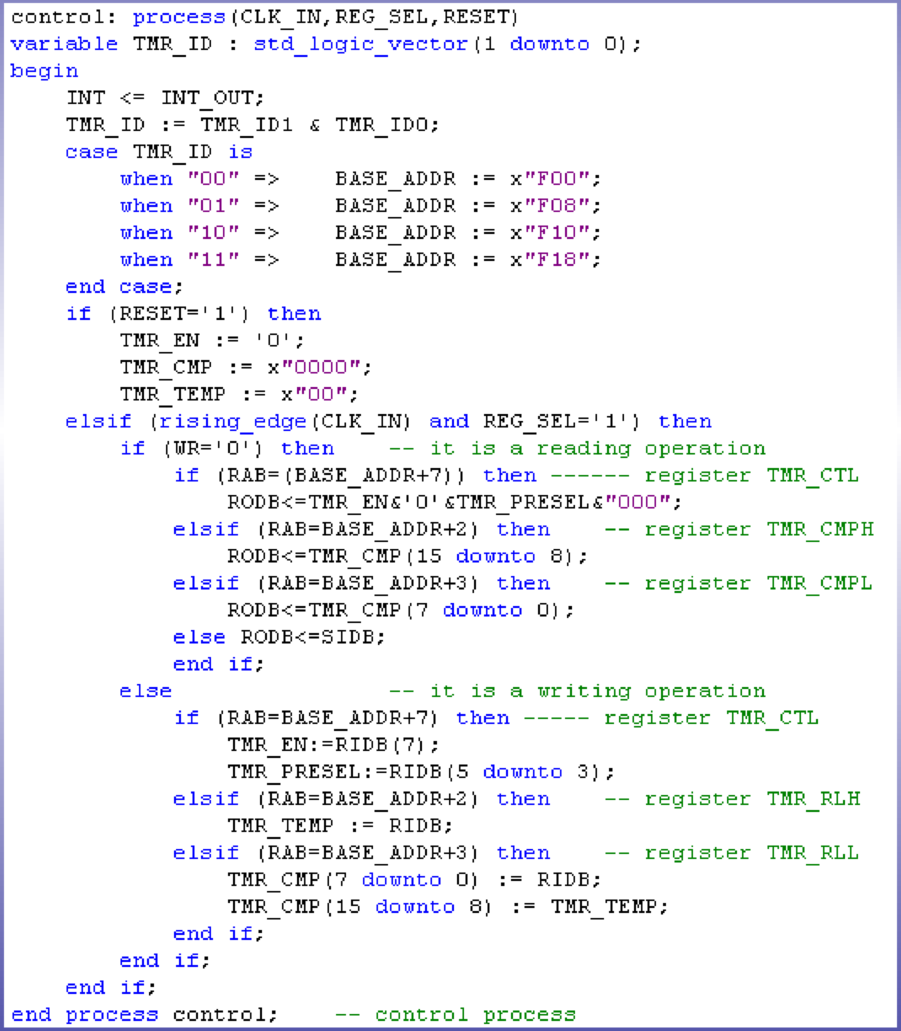


FPz8 – Timer Básico



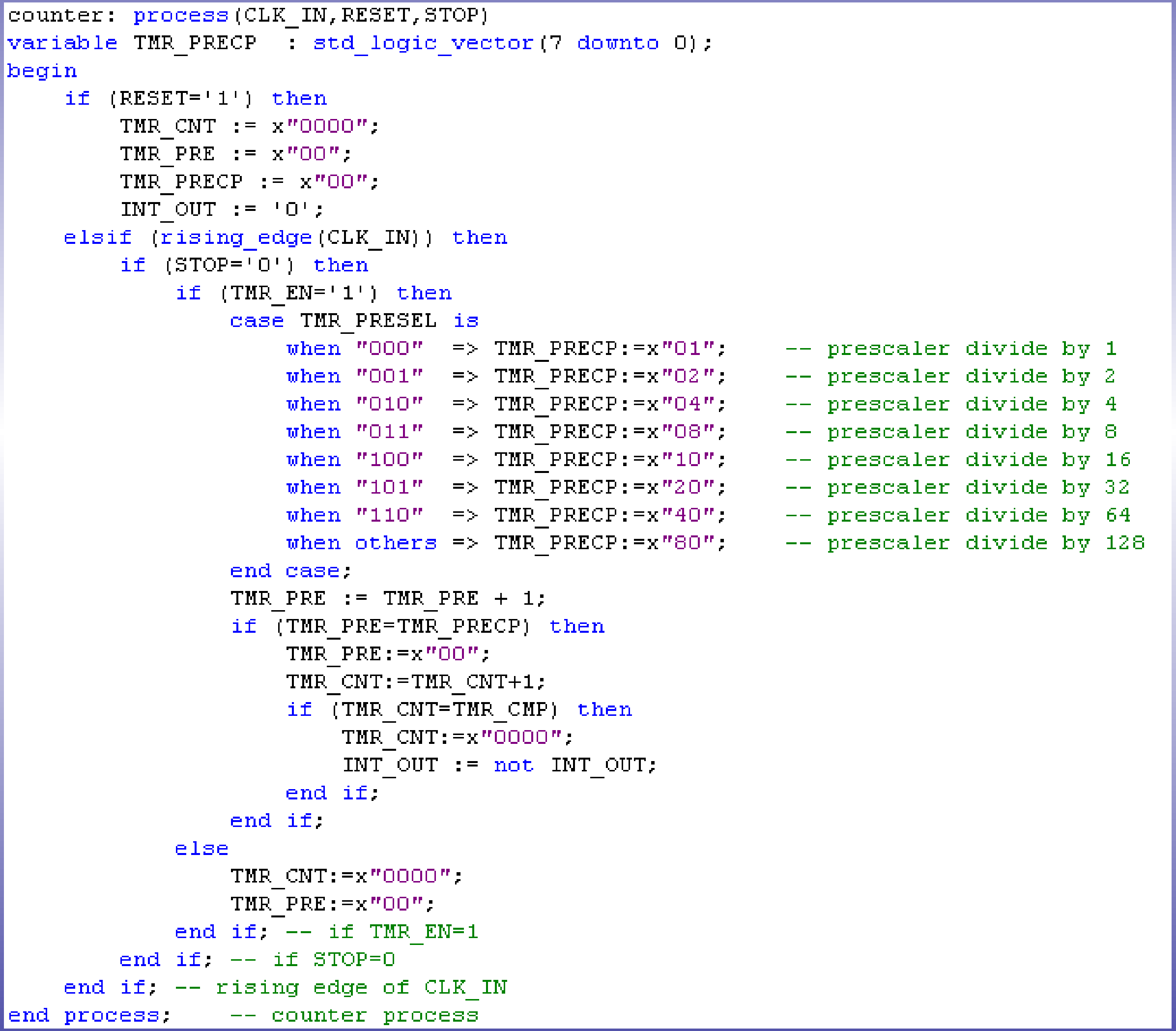
```
architecture timer of fpz8_timer is
shared variable TMR_EN      : std_logic;
shared variable TMR_CMP     : std_logic_vector(15 downto 0);
shared variable TMR_TEMP    : std_logic_vector(7 downto 0);
shared variable TMR_PRESEL  : std_logic_vector(2 downto 0);
shared variable TMR_CNT     : std_logic_vector(15 downto 0);
shared variable TMR_PRE     : std_logic_vector(7 downto 0);
shared variable INT_OUT     : std_logic;
shared variable BASE_ADDR   : std_logic_vector(11 downto 0);
begin
    control: process(CLK_IN,REG_SEL,RESET)
    counter: process(CLK_IN,RESET,STOP)
end timer;
```

FPz8 – Timer Básico



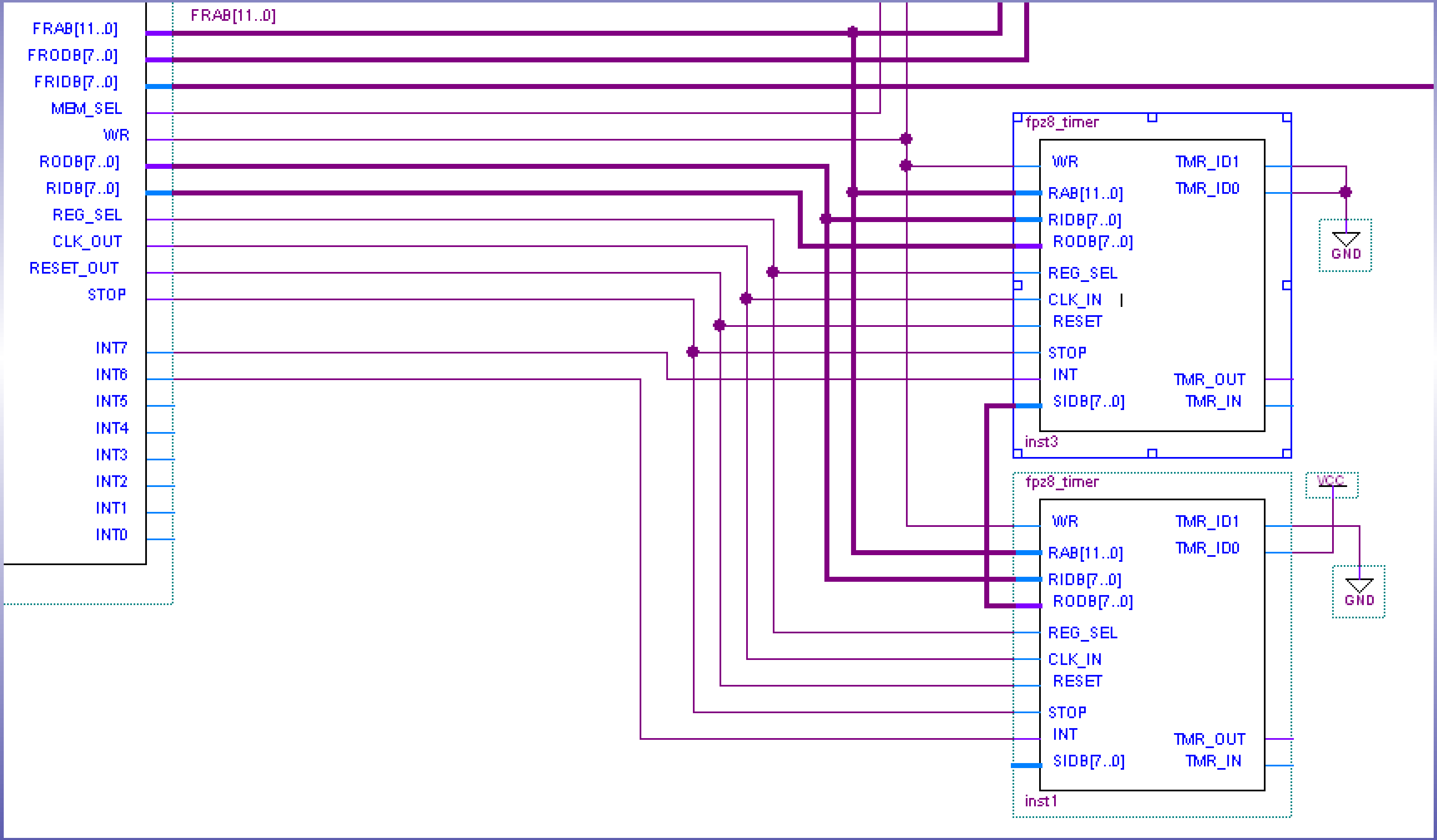
```vhdl
control: process(CLK_IN,REG_SEL,RESET)
variable TMR_ID : std_logic_vector(1 downto 0);
begin
    INT <= INT_OUT;
    TMR_ID := TMR_ID1 & TMR_ID0;
    case TMR_ID is
        when "00" =>    BASE_ADDR := x"F00";
        when "01" =>    BASE_ADDR := x"F08";
        when "10" =>    BASE_ADDR := x"F10";
        when "11" =>    BASE_ADDR := x"F18";
    end case;
    if (RESET='1') then
        TMR_EN := '0';
        TMR_CMP := x"0000";
        TMR_TEMP := x"00";
    elsif (rising_edge(CLK_IN) and REG_SEL='1') then
        if (WR='0') then    -- it is a reading operation
            if (RAB=(BASE_ADDR+7)) then ------ register TMR_CTL
                RODB<=TMR_EN&'0'&TMR_PRESEL&"000";
            elsif (RAB=BASE_ADDR+2) then    -- register TMR_CMPH
                RODB<=TMR_CMP(15 downto 8);
            elsif (RAB=BASE_ADDR+3) then    -- register TMR_CMPL
                RODB<=TMR_CMP(7 downto 0);
            else RODB<=SIDB;
            end if;
        else                -- it is a writing operation
            if (RAB=BASE_ADDR+7) then ----- register TMR_CTL
                TMR_EN:=RIDB(7);
                TMR_PRESEL:=RIDB(5 downto 3);
            elsif (RAB=BASE_ADDR+2) then    -- register TMR_RLH
                TMR_TEMP := RIDB;
            elsif (RAB=BASE_ADDR+3) then    -- register TMR_RLL
                TMR_CMP(7 downto 0) := RIDB;
                TMR_CMP(15 downto 8) := TMR_TEMP;
            end if;
        end if;
    end if;
end process control;    -- control process
```

FPz8 – Timer Básico



```
counter: process(CLK_IN,RESET,STOP)
variable TMR_PRECP  : std_logic_vector(7 downto 0);
begin
    if (RESET='1') then
        TMR_CNT := x"0000";
        TMR_PRE := x"00";
        TMR_PRECP := x"00";
        INT_OUT := '0';
    elsif (rising_edge(CLK_IN)) then
        if (STOP='0') then
            if (TMR_EN='1') then
                case TMR_PRESEL is
                    when "000"  => TMR_PRECP:=x"01";    -- prescaler divide by 1
                    when "001"  => TMR_PRECP:=x"02";    -- prescaler divide by 2
                    when "010"  => TMR_PRECP:=x"04";    -- prescaler divide by 4
                    when "011"  => TMR_PRECP:=x"08";    -- prescaler divide by 8
                    when "100"  => TMR_PRECP:=x"10";    -- prescaler divide by 16
                    when "101"  => TMR_PRECP:=x"20";    -- prescaler divide by 32
                    when "110"  => TMR_PRECP:=x"40";    -- prescaler divide by 64
                    when others => TMR_PRECP:=x"80";    -- prescaler divide by 128
                end case;
                TMR_PRE := TMR_PRE + 1;
                if (TMR_PRE=TMR_PRECP) then
                    TMR_PRE:=x"00";
                    TMR_CNT:=TMR_CNT+1;
                    if (TMR_CNT=TMR_CMP) then
                        TMR_CNT:=x"0000";
                        INT_OUT := not INT_OUT;
                    end if;
                end if;
            else
                TMR_CNT:=x"0000";
                TMR_PRE:=x"00";
            end if; -- if TMR_EN=1
        end if; -- if STOP=0
    end if; -- rising edge of CLK_IN
end process;    -- counter process
```

FPz8 – Timer Básico



Links

- www.zilog.com
- www.sctec.com.br/blog
- https://github.com/fabiopjve
- http://opencores.org/project,fpz8

Possibilidades Futuras

- Expandir o conjunto de instruções (página 2 dos opcodes)
- Implementar sistema de interleaving nos bancos de memória
- Reescrever o core utilizando micro-código e pipeline

Perguntas?