

# The GUS-16 V8 CPU

Jesús Arias

## 1 The GUS-16 V8 changes

Version	Arhitecture	Instr. Set	Comments	Uses
V1	Harvard	V1	Starting design, 2 memory spaces	Emulated
V2	Von Neumman	V1	2-stage pipeline, one memory space	Emulated
V3		V1+RETI	Interrupt Support	FPGA demo
V4		V4	LDPC replaces LDH	FPGA demo: GUSY
V5		V5	LD, ST with literal displacements	GUSY, Floppyton-GUS
V6		V6	new instruction set, multibit rotation	Floppyton-GUS v6
V7		V7	ROR instead of BIC	
V8		V8	Byte addresses, LDB / STB	

The V8 revision of the core has some important changes, namely:

- Byte addresses. The address space is now 65536 bytes, not words. This means:
  - The address bus lacks A[0]
  - The bit #0 of the PC register is always 0. PC is incremented by 2 when fetching instructions.
  - Memory displacements are accounted as bytes, both for loads, stores, and jumps.
- New load byte (LDB) and load byte signed (LDBS) instructions.
- New store byte (STB) instruction. When executed only the content of the upper or lower halves of the data bus have to be written into memory. The core includes two new signals to deal with byte writes:
  - BHE: Bus high enable. Read / write D[15:8]
  - BLE: Bus low enable. Read / write D[7:0]
- Little-Endian architecture: LSB bytes on even addresses.
- Load and store instructions now have a signed 5-bit displacement. This means a byte can be accessed from 16 positions below the base register up to 15 positions above the base register (14 positions in the case of 16-bit words)
- Word accesses on odd addresses will result in the upper and lower bytes of the data being swapped. No exception will be generated (ARM-like).

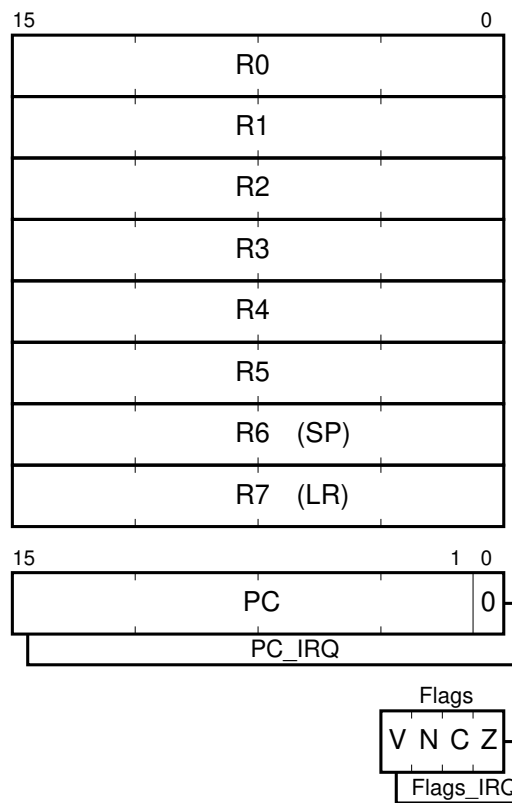


Figure 1: GUS16-V8 register set

## 2 GUS16-V8 features

### 2.1 Programmer's model

The GUS16 processor has 9 registers available to programmers in addition to four condition code flags. These registers are:

- R0 to R7. General purpose working registers, 16-bit wide. In the older GUS16 versions all registers were equivalent, but in the new “v8” version the register R7 has a hardware defined function: It acts as a “link register”, meaning it is written with the return address when calling a subroutine via the “JAL” (Jump And Link) or “JMPL” (Jump absolute and Link) instructions. The remaining registers are truly equivalent, yet, R6 is the register usually acting as the system’s stack pointer.
- PC. The program counter is in fact two registers: The normal PC is used when executing the main program while the alternative PC register is switched in when entering an interrupt routine. The PC is switched back to its normal mode register after executing the “RETI” (Return from Interrupt) instruction. The normal mode PC is the only register with a known initial value after reset (0x0000). The initial value of the interrupt mode PC depends on the “ivector” inputs of the core, resulting in the value (VECTORBASE + ivector\*4), with VECTORBASE defaulting to 0x0000. The PC has its bit #0 fixed at zero, meaning it is always pointing to an even address in memory.
- Flags. These are four bits that reflects the results of previous instructions and are checked when executing conditional jumps. And, in line with the PC, there are two flag registers: one for normal mode and other for interrupt routines. Only some instructions change these flags, and some can change Z and N but not C and V. The flags are:
  - Z. Zero. It gets set when a result is zero. Notice that the Load instruction also writes this flag.

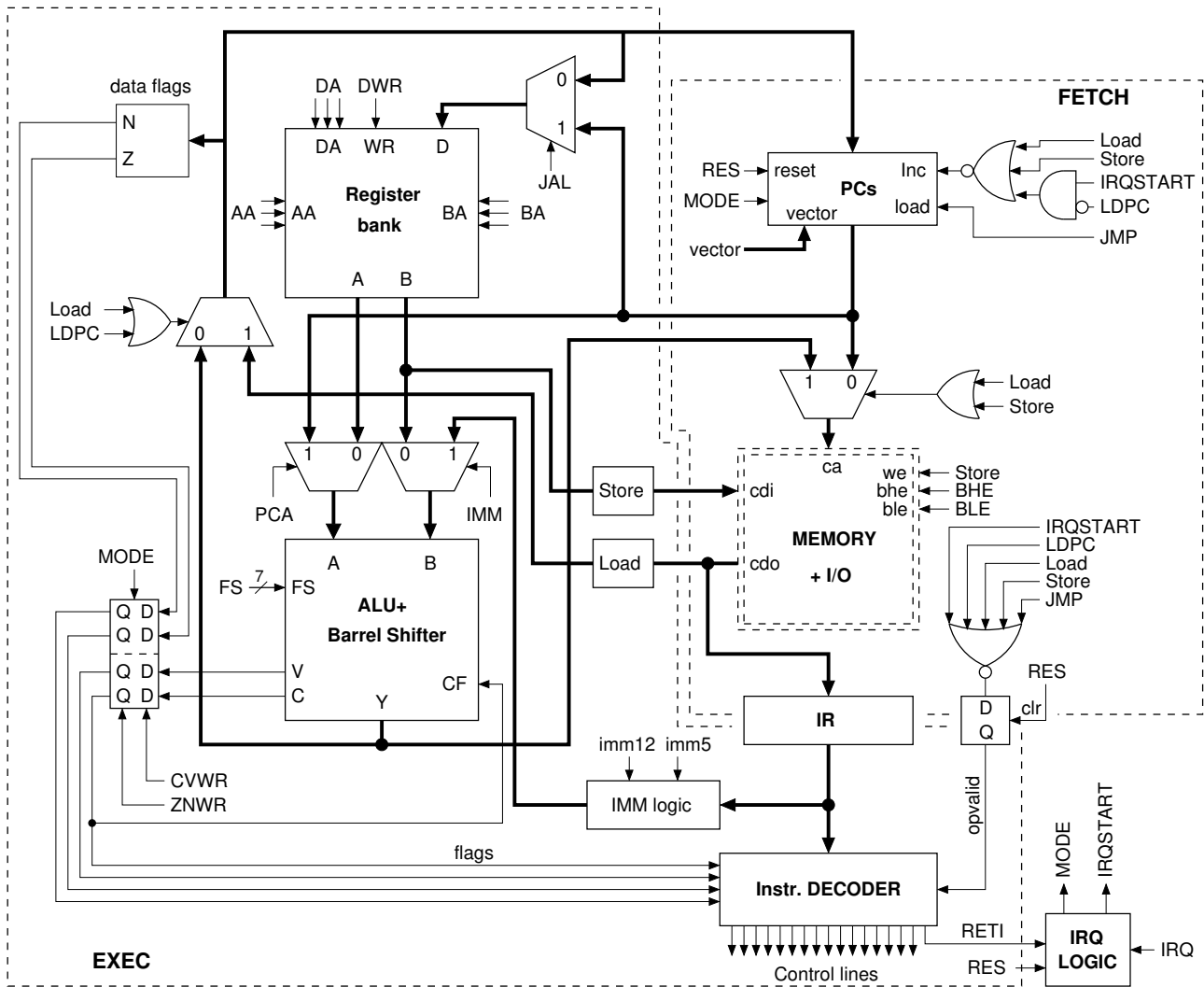
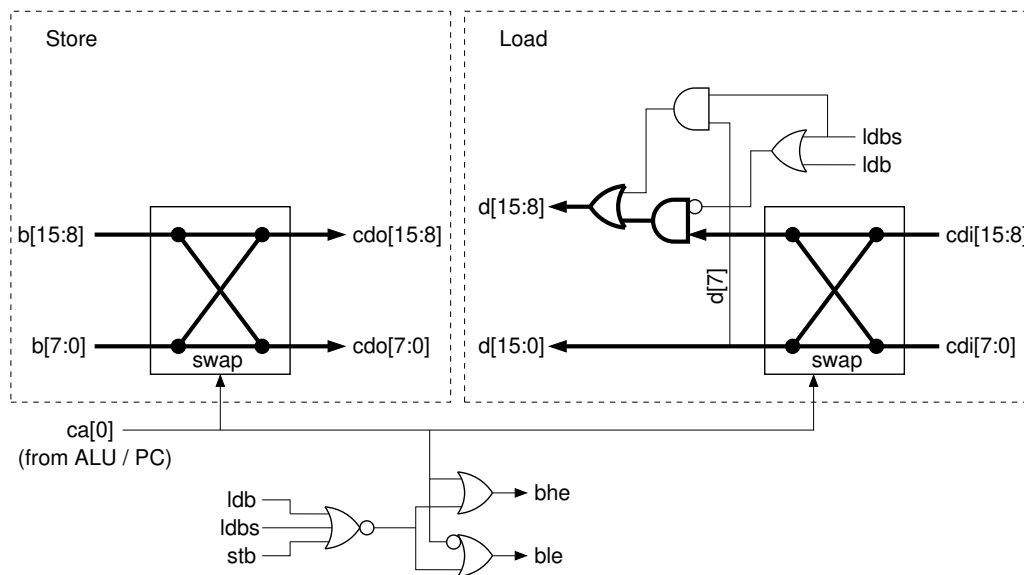


Figure 2: GUS16-V8 Internal blocks

- C. Carry. It gets set when the result of an arithmetic instruction doesn't fit in 16 bits. Notice that this flag is an inverted "borrow" for subtractions and this means it gets set if there is no "borrow" in the subtraction. It can also be written with the bit #0 of a general purpose register when executing shifts and rotations to the right. Its value can affect not only the conditional jumps, but also the "arithmetic with carry" instructions and RORC (rotate right through carry)
- N. Negative. It is just a copy of the bit #15 of the result. It gets set if the result was a negative value for the "two's complement" arithmetic. Notice that the Load instruction also writes this flag.
- V. Overflow. It gets set when there is an unexpected change in the sign of the results, and it only makes sense when dealing with signed variables (use the carry flag instead for unsigned values)

## 2.2 Internal architecture

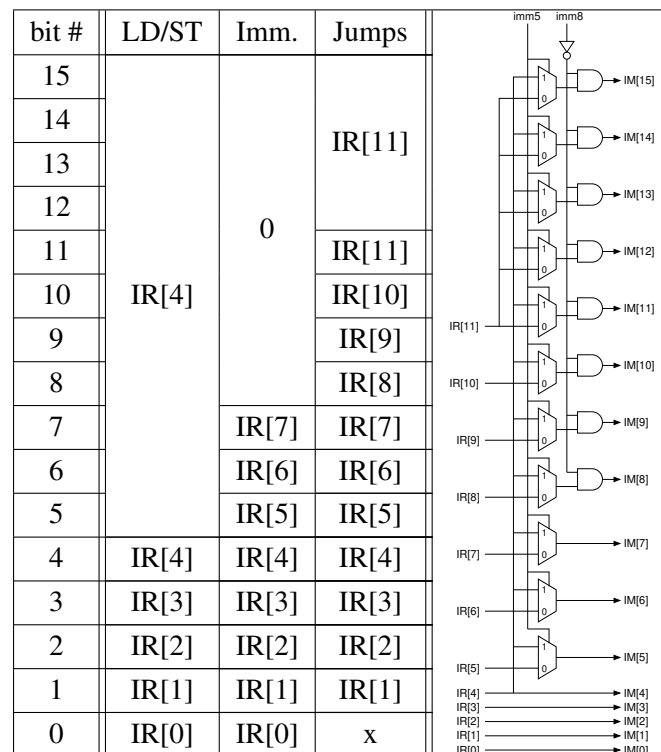
The block diagram of the GUS16-V8 core is presented in figure 2. This figure is almost identical to the V6 and V7 revisions of the core, except for the "Load" and "Store" blocks that are now present in series with the memory data buses. Of course, there are many other differences inside the other blocks, like "PCs", "IMM logic", and "Instr. Decoder", but let's first present the new Load and Store blocks. These blocks have to deal with the manipulation of bytes, and their internal diagrams are:



For the case of “Store” instructions the output bytes have to be swapped if an odd address is used as the memory pointer. This also have to be done in the case of “Load” instructions, but here more processing is still needed for the high byte, d[15:8]. This byte has to be forced as zero for the “load byte”, LDB, instruction, while for the “load byte signed”, LDBS, instruction the high byte is zero or 0xFF depending on the sign bit of the loaded byte (d[7]).

Also, I want to remark that the signals “bhe” and “ble” are mandatory for stores but optional for loads, as memory reads can be always 16-bit wide but only the desired byte is actually loaded into the core.

Another block with notable changes is the one that presents the literal constants as data for the ALU (IMM logic). Now the 5-bit displacements for load and store instructions is a signed field, so, the block output depending on the instruction being executed is:



Notice that the bit #0 is irrelevant for jumps because the PC has its bit #0 fixed at zero. In summary, the lower 5 bits are the same as IR's, the next 3 bits can have two different values, and the 8 MSB can have three different values, but one of them is zero (11 Logic cells).

## 2.3 Instruction Set

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	mnemonic	flags	operation	
0	0	0	0	0												ADD	C V N Z	RD= RA + RB	
0	0	0	0	0												SUB	C V N Z	RD= RA – RB	
0	0	0	0	0												ADC	C V N Z	RD= RA + RB+Cflag	
0	0	0	0	0												SBC	C V N Z	RD= RA– RB –~cflag	
0	0	0	0	0	1											AND	– – N Z	RD= RA & RB	
0	0	0	0	0	1											OR	– – N Z	RD= RA   RB	
0	0	0	0	0	1											XOR	– – N Z	RD= RA ^ RB	
0	0	0	0	0	1											ROR	– – N Z	RD = (RB>>RA)   (RB<<16–RA)	
0	0	0	0	1	0	–	–	–								TST	– – N Z	RA & RB	
0	0	0	0	1	0	–	–	–								CMP	C V N Z	RA – RB	
0	0	0	0	1	1														
0	0	0	1	0	0											ADDI	C V N Z	RD= RD + ulit8	
0	0	0	1	0	1											SUBI	C V N Z	RD= RD – ulit8	
0	0	0	1	1	0											ADCI	C V N Z	RD= RD + ulit8 +Cflag	
0	0	0	1	1	1											SBCI	C V N Z	RD= RD – ulit8 – ~Cflag	
0	0	1	0	0	0											ANDI	– – N Z	RD= RD & ulit8	
0	0	1	0	0	1											ORI	– – N Z	RD= RD   ulit8	
0	0	1	0	1	0											XORI	– – N Z	RD= RD ^ ulit8	
0	0	1	0	1	1											CMPI	C V N Z	RD – ulit8	
0	0	1	1	0	0											TSTI	– – N Z	RD & ulit8	
0	0	1	1	0	1											LDI	– – – –	RD= ulit8	
0	0	1	1	1	0														
0	0	1	1	1	1		RD	0	ulit4h		RB		ulit4l			RORI	– – N Z	RD = (RB>>uli4)   (RB<<16–ulit4)	
0	0	1	1	1	1		RD	1	0	0		RB		0	0	RORC	C ? N Z	RD = {Cflag,RB>>1}, Cflag=RB0	
0	0	1	1	1	1		RD	1	0	0		RB		0	1	SHR	C ? N Z	RD = RB>>1, Cflag=RB0	
0	0	1	1	1	1		RD	1	0	0		RB		1	0	SHRA	C ? N Z	RD = RB>>1 (signed) , Cflag=RB0	
0	0	1	1	1	1		RD	1	0	1		RB		0	0	NOT	– – N Z	RD = ~RB	
0	0	1	1	1	1		RD	1	0	1		RB		0	1	NEG	C V N Z	RD = –RB	
0	0	1	1	1	1		RD	1	0	1		RB		1	0	MOV	– – – –	RD = RB	
0	0	1	1	1	1		RD	1	1	1	–	0	0	0	0	LDPC class	LI	– – – –	RD = Mem(PC++)
0	0	1	1	1	1	–	–	–	1	1	1	–	0	1	0		JMP	– – – –	PC = Mem(PC)
0	0	1	1	1	1	–	–	–	1	1	1	–	1	0	0		JMPL	– – – –	PC = Mem(PC), Rlink=PC
0	0	1	1	1	1	–	–	–	1	1	1	0	0	0	0	1	NOP *	– – – –	–
0	0	1	1	1	1	–	–	–	1	1	1	–	0	1	0	1	WFI	– – – –	wait for interrupt
0	0	1	1	1	1	–	–	–	1	1	1	–	1	0	0	1	BRK	– – – –	breakpoint
0	0	1	1	1	1	–	–	–	1	1	1		RB	1	0	JIND	– – – –	PC = RB	
0	0	1	1	1	1	–	–	–	1	1	1	–	–	–	1	1	RETI	– – – –	return from interrupt
1	0	0	0	0	0		RD				RA	sdisp5				LD	– – N Z	RD= Mem(RA+sdisp5) (16)	
1	0	0	0	0	1		RD				RA	sdisp5				LDB	– – N Z	RD= Mem(RA+sdisp5) (u8)	
1	0	0	0	1	0														
1	0	0	0	1	1		RD				RA	sdisp5				LDBS	– – N Z	RD= Mem(RA+sdisp5) (s8)	
1	0	0	1	0	0		RB				RA	sdisp5				ST	– – – –	Mem(RA+sdisp5)=RB (16)	
1	0	0	1	0	1		RB				RA	sdisp5				STB	– – – –	Mem(RA+sdisp5)=RB (8)	
1	0	0	1	1		sdisp11									0	JR	– – – –	PC = PC+sdisp11	
1	0	0	1	1		sdisp11									1	JAL	– – – –	PC = PC+sdisp11, Rlink=PC	
1	0	0	1	0	0	sdisp11									0	JZ	– – – –	PC = PC+sdisp11 if Zflag	
1	0	0	1	0	0	sdisp11									1	JNZ	– – – –	PC = PC+sdisp11 if ~Zflag	
1	0	0	1	0	1	sdisp11									0	JC	– – – –	PC = PC+sdisp11 if Cflag	
1	0	0	1	0	1	sdisp11									1	JNC	– – – –	PC = PC+sdisp11 if ~Cflag	
1	0	0	1	1	0	sdisp11									0	JMI	– – – –	PC = PC+sdisp11 if Nflag	
1	0	0	1	1	0	sdisp11									1	JPL	– – – –	PC = PC+sdisp11 if ~Nflag	
1	0	0	1	1	1	sdisp11									0	JV	– – – –	PC = PC+sdisp11 if Vflag	
1	0	0	1	1	1	sdisp11									1	JNV	– – – –	PC = PC+sdisp11 if ~Vflag	

In the above table the instruction set of the GUS16-V8 CPU is presented with the new instructions marked in red. Notice that the instructions of the 'LDPC' class have an effective 32 bit opcode because they all use the PC as a data pointer and they have to be followed by a 16-bit value. The new instructions with respect to the V7 version are:

- **TST Ra, Rb** : Test. It performs the logic AND between the two registers and updates the Z and N flags, but the result is discarded.
- **CMP Ra, Rb** : Comparison. It computes Ra - Rb and updates all the flags, but the result is discarded.
- **TSTI Rd, nn** : Test bit with immediate. It performs the AND between the register and the 8-bit literal and updates the Z and N flags, but the result is discarded.
- **MOV Rd, Rb** : Move registers. No flags are changed.
- **LI Rd, imm16** : Load 16-bit immediate (called 'LDPC' before). It has to be followed with a 16-bit word.
- **JMP imm16** : Absolute jump. It has to be followed with a 16-bit word whose value is loaded into the PC.
- **JMPL imm16** : Absolute jump and link. Does the same jump as JMP but it also stores the address of the next instruction (the current value of PC plus 2) into LR (R7).
- **NOP** : No operation. It changes nothing, neither the registers Rx nor the flags. Notice that now all unused op-codes are executed as NOPs as a safety measure, but this particular one will be preserved for future revisions.
- **WFI** : Wait for interrupt. Halts the execution until an interrupt is requested. It also reduces the power consumption of the core. (IDLE mode)
- **BRK** : Breakpoint. Used in emulators to stop the execution. The core executes it as a NOP.
- **LDB Rd,(Ra+sdisp)** : Load byte. The 8 MSB bits of Rd are written with zeroes.
- **LDBS Rd,(Ra+sdisp)** : Load byte signed. The 8 MSB bits of Rd are written with the value of Rd[7]
- **STB (Ra+sdisp),Rb** : Store byte.
- **JNV** : Jump relative if not overflow. This condition wasn't supported on previous core revisions.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	mnemonic	flags	operation
0	1	1	1	1			RD	1	1	0	—	—	—	0	0	<b>RDPC</b>	— — — —	RD = PC_normal
0	1	1	1	1			RD	1	1	0	—	—	—	0	1	<b>RDFL</b>	— — — —	RD = Flags_normal (VNCZ)
0	1	1	1	1		—	—	—	1	1	0		RB	1	0	<b>WRPC</b>	— — — —	NOP / PC_normal = RB
0	1	1	1	1		—	—	—	1	1	0		RB	1	1	<b>WRFL</b>	RB[3:0]	Flags_normal = RB
0	1	1	1	1			ulit3	1	1	1	—	1	1	0	1	<b>SWI</b>	— — — —	software interrupt

In the above table an instruction set extension is also presented. These instructions are mainly intended for multitasking and debugging, and they are:

- **RDPC Rd** : Read PC. The value of the normal mode PC register is copied to Rd. Notice that the PC register for the interrupt mode can't be read with this instruction.

- **RDFL Rd** : Read Flags. The normal mode flag register is copied to the lower bits of Rd along with the actual mode of the core (0: normal, 1: interrupt) according to the following table. The interrupt mode flags can't be read.

Rd[15:5]	Rd[4]	Rd[3]	Rd[2]	Rd[1]	Rd[0]
0	Mode	V	N	C	Z

- **WRPC Rb** : Write PC. In normal mode this instruction does nothing (NOP), but in interrupt mode the value of Rb is written into the normal mode PC register, thus changing the return address for RETI.
- **WRFL Rb** : Write Flags. The lower 4 bits of Rb are written into the normal mode flag register. This instruction can be executed in interrupt or normal modes but only the normal mode flags are changed. The bit order is:

V	N	C	Z
Rb[3]	Rb[2]	Rb[1]	Rb[0]

- **SWI nn** : Software interrupt. Changes the core to interrupt mode and jumps to the interrupt routine with the vector stated in its 3-bit literal (jumps to VECTORBASE + nn\*4).

## 2.4 Interface

These are the signals of the GUS16 V8 core:

```

module CPU_GUSV8(
output [15:1]ca,    // Core Address
output [15:0]cdo,   // Core Data Output
output we,         // Write Enable (store)
output bhe,        // Bus High Enable
output ble,        // Bus Low Enable
input  [15:0]cdi,   // Core Data Input
input  clk,        // Clock
input  cen,        // Clock enable
input  reset,      // async. Reset (active high)
input  irq,        // Interrupt Request
input  [2:0]ivector,// Interrupt vector
output reg mode,    // CPU mode 0: normal 1: Interrupt
output fetch,      // Valid opcode fetch if 1
output dload,      // Data load if 1
input  brk        // Break interrupt if 1
);

```

- **'ca'**: Address Bus. It lacks ca[0] because the external memory has to be 16-bit wide and addresses are for bytes.
- **'cdo' / 'cdi'**: Data buses (output and input)
- **'we'**: Write enable. Active high during the execution cycle of Store instructions.
- **'bhe' / 'ble'**: Byte high and byte low enable, active high. When high, and 'we' active, the corresponding half of the output data bus (cdo[15:8] / cdo[7:0]) has to be written to the external memory. These signals are also active for byte reads, yet, they aren't strictly required (the data present at the unselected 'cdi' half will be ignored anyway). Both signals are active simultaneously during 16-bit loads, stores, and opcode fetches.
- **'clk'**: Core clock. Active on rising edges.

- **'cen'**: Clock enable. Synchronous input that pauses the core if low during the rising edge of 'clk' (wait state).
- **'reset'**: Reset. Asynchronous input that resets the core if high (makes PC=0x0000, mode=normal, and invalidates last opcode)
- **'irq'**: Interrupt request. Active high. It must stay high at least two clock cycles before the core changes to interrupt mode and jumps of the related interrupt routine. Also, this signal is ignored during the execution of interrupt routines except for their last RETI instruction (chained interrupts if 'irq' still active, return to normal mode otherwise).
- **'ivector'**: A 3-bit input that identifies the current interrupt source. If 'irq' is active the core changes to interrupt mode and jumps to the address 'VECTORBASE' + 'ivector'  $\times$  4. 'VECTORBASE' is a core parameter that defaults to 0x0000. Be aware that in this case an interrupt with 'ivector'=0 jumps to the same address as a hardware reset.
- **'mode'**: Core mode output. Low: normal, high: interrupt.
- **'fetch'**: Valid opcode fetch output. High on opcode fetches, low during loads, stores, or taken jumps. This output, along with 'mode', 'we', and 'dload', provides support for hardware debug.
- **'dload'**: Data Load. High during the execution of Load instructions.
- **'brk'**: Breakpoint input. When high the core changes to interrupt mode and jumps to interrupt vector #7. Set this input low if unused. This signal looks like another interrupt input, but it has some important differences with 'irq':
  - When 'brk' is active the core changes to interrupt mode on the next clock cycle while the 'irq' input takes two clock cycles.
  - 'brk' don't execute the current instruction (pointed by the normal mode PC) while 'irq' finishes the current instruction before jumping to the interrupt routine.
  - 'brk' is inhibited one cycle after the core leaves the interrupt mode. This allows the opcode at the breakpoint address to be executed after the interrupt.
  - 'brk' has the implicit vector #7, and the highest priority.

### 3 Source listing



```

//-----
//-----
// CPU GUS16 v8 (new instruction set)                                Jesús Arias (2023-2026)
//-----
//-----
module CPU_GUSV8(
output [15:1]ca,      // Core Address
output [15:0]cdo,     // Core Data Output
output we,           // Write Enable (store)
output bhe,          // Bus High Enable
output ble,          // Bus Low Enable
input [15:0]cdi,     // Core Data Input
input clk,           // Clock
input cen,           // Clock enable
input reset,         // async. Reset (active high)
input irq,           // Interrupt Request
input [2:0]ivector,  // Interrupt vector
output reg mode,     // CPU mode 0: normal 1: Interrupt
output fetch,        // Valid opcode fetch if 1
output dload,        // Data load if 1
input brk            // Break interrupt if 1
);

parameter VECTORBASE = 16'h0000;
parameter REGLINK = 3'd7;

//----- ALIAS / state -----
assign we = store;
assign dload = load;
assign fetch = ~(load | ldpc | store | jump);

//----- CLOCK -----
wire clken = cen & (irq | ~wfi);

//-----
//----- BUSES -----
//-----
assign ca = (load | store) ? aluf[15:1] : regpc[15:1]; // Address bus
assign cdo = store & aluf[0] ? {regb[7:0],regb[15:8]} : regb;
wire [15:0]cdix = (ldb | ldbs) & aluf[0] ? {cdi[7:0],cdi[15:8]} : cdi;
wire [15:0]cdiy;
assign cdiy[7:0] = cdix[7:0];
assign cdiy[15:8] = ldbs ? {8{cdix[7]}} : (ldb ? 8'h00 : cdix[15:8]);

assign bhe = (ldb | ldbs | stb) ? aluf[0] : 1'b1;
assign ble = (ldb | ldbs | stb) ? ~aluf[0] : 1'b1;

//-----
//----- REGISTER BANK -----
//-----

wire [15:0]rega;      // output A
wire [15:0]regb;      // output B
wire [2:0]aa;         // A address
wire [2:0]ba;         // B address
wire [2:0]da;         // Dest address
wire dwr;             // Write register if 1

reg [15:0]regs[0:7];
// register write and JMPL return address adjustment
always @(posedge clk)
    if (clken) if (dwr) regs[da]<= (jal | jmpl ? regpc : busd) + {jmpl,1'b0};
assign rega=regs[aa];
assign regb=regs[ba];

`ifdef SIMULATION
// for debug
wire [15:0]R0=regs[0];
wire [15:0]R1=regs[1];

```

```

wire [15:0]R2=regs[2];
wire [15:0]R3=regs[3];
wire [15:0]R4=regs[4];
wire [15:0]R5=regs[5];
wire [15:0]R6=regs[6];
wire [15:0]R7=regs[7];
wire [15:0]PC0={PC[0],1'b0};
`endif
//-----
// ----- PC -----
//-----

wire [15:0]regpc;
wire pcinc,jump,irqstart;
wire [15:1]vector;
wire [2:0]svector = brkrq ? 3'b111 : (swi ? IR[10:8] : ivector);

assign vector = (VECTORBASE>>1)+{svector,1'b0}; // IRQ address
assign pcinc = ~(load | store | (irqstart&(~ldpc))); // Increment PC

wire [15:1]pcnxt= jump ? busd[15:1] : regpc[15:1] + pcinc; // next value for PC (main /
IRQ)

reg [15:1]PC [0:1];
always @(posedge clk or posedge reset )
    if (reset) PC[0]<=15'h0000;
    else if (clken) PC[0]<= mode ? (wrspc & ~IR[0] ? regb[15:1] : PC[0] ): pcnxt;
// registro PC modo IRQ
always @(posedge clk ) if (clken)
    PC[1]<= (mode&(~reti)) ? pcnxt : vector;

assign regpc={PC[mode],1'b0};

//-----
// ALU
//-----
wire [15:0]alua;
wire [15:0]alub;
wire cd,vd,zd,nd;
wire [1:0]aluop;

assign alua = pca ? regpc : rega;
assign alub = (imm ? busimm : regb);

wire c0= xa ? cf : ib; // input carry
// data inputs to adder
wire [15:0]sa= za ? 16'b0 : alua; // Zero input A if za is high
wire [15:0]sb= ib ? (~alub) : alub; // invert input B if ib is high

// Basic ALU
// change codes until minimum LCs achieved

`ifdef OPTIMIZE
    `include "aluop.v"
`else
    localparam ADD = 2'd1;
    localparam AND = 2'd2;
    localparam _OR = 2'd0;
    localparam XOR = 2'd3;
`endif
reg [15:0]aluf; // not regs
reg c15;
always@*
    case (aluop)
        ADD : {c15,aluf} = sa+sb+c0; // ADD
        XOR : {c15,aluf} = {1'bx,sa ^ sb}; // XOR
        _OR : {c15,aluf} = {1'bx,sa | sb}; // OR
        AND : {c15,aluf} = {1'bx,sa & sb}; // AND
    endcase

```

```

// Overflow flag
assign vd = ((~sb[15])&(~sa[15])&aluf[15]) | (sb[15]&sa[15]&(~aluf[15]));

//----- BARREL shifter -----

wire [3:0]bsi0mux = {cf,aluf[15],1'b0,aluf[0]}; // LSB select
wire bsi0 = bsi0mux[rorssel];
wire [15:0]bsi = {aluf[15:1],bsi0};

wire [3:0]bssel = rori ? {IR[6:5],IR[1:0]} : (ror ? alua[3:0]:{3'b0,|rorssel});

wire [15:0]bsl1 = bssel[3] ? { bsi[7:0], bsi[15:8]} : bsi;
wire [15:0]bsl2 = bssel[2] ? {bsl1[3:0],bsl1[15:4]} : bsl1;
wire [15:0]bsl3 = bssel[1] ? {bsl2[1:0],bsl2[15:2]} : bsl2;
wire [15:0]y = bssel[0] ? { bsl3[0],bsl3[15:1]} : bsl3;

// result data selection
reg [15:0]busd; // input D (destination), not a register
always@*
  case ( {load | ldpc, rdspc, IR[0]})
    3'b1xx: busd <= cdi; // LOAD or LDPC
    3'bx10: busd <= {PC[0],1'b0}; // RDPC
    3'bx11: busd <= {11'h000,mode,cv[0][0],zn[0][0],cv[0][1],zn[0][1]}; // RDFL
    default: busd <= y; // ALU result
  endcase

// Carry flag
assign cd = (|rorssel) ? alub[0] : c15;
// Flags Z, N
assign zd = (busd==0);
assign nd = busd[15];

//-----
//----- Flag register -----
//-----

wire cf,vf,zf,nf;
wire wrcv,wrzn;
reg [1:0]zn[0:1];
reg [1:0]cv[0:1];
assign {zf,nf}=zn[mode];
assign {cf,vf}=cv[mode];
always @(posedge clk) if (clken) begin
  zn[0]<= wrzn & ~mode ? {zd,nd} : ( wrspc & IR[0] ? {regb[0],regb[2]} : zn[0]);
  zn[1]<= wrzn & mode ? {zd,nd} : zn[1];
  cv[0]<= wrcv & ~mode ? {cd,vd} : ( wrspc & IR[0] ? {regb[1],regb[3]} : cv[0]);
  cv[1]<= wrcv & mode ? {cd,vd} : cv[1];
end

//-----
// Instruction register
//-----
wire opvali= ~(load | ldpc | store | jump | irqstart); // next op-code valid

reg [15:0]IR;
reg opval=0; // current op-code valid
always @(posedge clk) if (clken) IR<= cdi;
always @(posedge clk or posedge reset)
  if (reset) opval<=1'b0;
  else if (clken) opval<= opvali;

//-----
// Immediate opperands
//-----
wire [15:0]busimm = (imm5 ? {{11{IR[4]}},IR[4:0]} : 16'h0000) |
  (imm12 ? {{4{IR[11]}},IR[11:0]} : 16'h0000) |
  (~imm5 & ~imm12 ? {8'h00,IR[7:0]} : 16'h0000);

```

```

//-----
// Interrupts
//-----

// delayed mode to avoid to trigger a second BRK on leaving IRQ7
reg omode;
always @(posedge clk) if (clken) omode <= mode;

// BRK disabled during the (extended) IRQ mode
wire brkrq = ~(omode | mode) & brk;

reg irqq0=0;
wire ireti=reti&(~irq);
always @(posedge clk or posedge reset)
begin
    if (reset) begin
        irqq0<=1'b0;
        mode<=1'b0;
    end
    else if (clken) begin
        irqq0 <= (~ireti) & (irqq0 | irq) | swi | brkrq;
        mode <= (~ireti) & irqq0 | swi | brkrq;
    end
end
assign irqstart = (~mode) & irqq0 | swi | brkrq;

//-----
// DECODING
//-----

// 8 to 1 mux for conditional jump
wire [7:0]ccond={~vf,vf,~nf,nf,~cf,cf,~zf,zf};
wire jcc_taken = jcc & ccond[{IR[13:12],IR[0]}];

// Some op-codes get decoded directly...
wire jcc = opval & (IR[15:14]==2'b11);
wire jind = opval & (IR[15:11]==5'b01111) & (IR[7:5]==3'b111) & IR[1];
wire jr = opval & (IR[15:12]==4'b1011) & (~IR[0]);
wire jal = opval & (IR[15:12]==4'b1011) & (IR[0]);

wire ror = opval & (IR[15:11]==5'b00001) & (IR[1:0]==2'b11);
wire rori = opval & (IR[15:11]==5'b01111) & (~IR[7]);
wire ldpc = opval & (IR[15:11]==5'b01111) & (IR[7:5]==3'b111) & (IR[1:0]==2'b00);
wire jmp = ldpc & IR[2];
wire jmp1 = ldpc & IR[3];
wire wfi = opval & (IR[15:11]==5'b01111) & (IR[7:5]==3'b111) & (IR[3:0]==4'b0101);
wire swi = opval & (IR[15:11]==5'b01111) & (IR[7:5]==3'b111) & (IR[3:0]==4'b1101);
wire reti = opval & (IR[15:11]==5'b01111) & (IR[7:5]==3'b111) & (IR[1:0]==2'b11);
wire ld = opval & (IR[15:11]==5'b10000);
wire ldb = opval & (IR[15:11]==5'b10001);
wire ldbs = opval & (IR[15:11]==5'b10011);
wire st = opval & (IR[15:11]==5'b10100);
wire stb = opval & (IR[15:11]==5'b10101);
wire load = (ld | ldb | ldbs);
wire store= (st | stb);

wire imm8 = opval & imm & ~(load | store | jal | jr | jcc);
wire imm12 = jr | jal | jcc;
wire imm5 = ld | ldb | ldbs | st | stb;

wire rdspc = opval & (IR[15:11]==5'b01111) & (IR[7:5]==3'b110) & (IR[1]==1'b0);
wire wrspc = opval & (IR[15:11]==5'b01111) & (IR[7:5]==3'b110) & (IR[1]==1'b1);

assign jump = jcc_taken | jr | jal | jind | jmp | jmp1;

// register bank selection
assign aa = imm8 ? IR[10:8] : IR[7:5];
assign ba = store ? IR[10:8] : IR[4:2];
assign da = jal | jmp1 ? REGLINK : IR[10:8];

```

```

// For making the decoding table easier to read
localparam _ = 1'b0;
localparam _1 = 1'b1;
localparam _x = 1'bx;
localparam _xx = 2'bx;
localparam _xx_ = 2'bx;
localparam _ = 2'b00;
localparam _ASR = 2'b01;
localparam ASRA = 2'b10;
localparam RORC = 2'b11;

reg [11:0] control; // combined control lines (not register)
wire za, ib, xa, imm, wd, wc, wz, pca; // single control lines
wire [1:0] rorsel; // shift select (0: none/RORI, 1: ASR, 2: ASRA, 3: RORC)
assign {aluop, za, ib, xa, rorsel, imm, wd, wc, wz, pca} = control;

always@*
  casex ({IR[15:11], IR[7:5], IR[1:0]})
    // alu za ib xa rors imm wd wc wz pca
    10'b00000_00: control <= {ADD, __, __, __, __, __, _1, _1, _1, __}; // ADD
    10'b00000_01: control <= {ADD, __, _1, __, __, __, _1, _1, _1, __}; // SUB
    10'b00000_10: control <= {ADD, __, __, _1, __, __, _1, _1, _1, __}; // ADC
    10'b00000_11: control <= {ADD, __, _1, _1, __, __, _1, _1, _1, __}; // SBC

    10'b00001_00: control <= {AND, __, __, _x, __, __, _1, __, _1, __}; // AND
    10'b00001_01: control <= {OR, __, __, _x, __, __, _1, __, _1, __}; // OR
    10'b00001_10: control <= {XOR, __, __, _x, __, __, _1, __, _1, __}; // XOR
    10'b00001_11: control <= {OR, _1, __, _x, __, __, _1, __, _1, _x}; // ROR

    10'b00010_00: control <= {AND, __, __, _x, __, __, __, __, _1, __}; // TST
    10'b00010_01: control <= {ADD, __, _1, __, __, __, __, __, _1, __}; // CMP

    10'b00100_00: control <= {ADD, __, __, __, __, __, _1, _1, _1, __}; // ADDI
    10'b00101_00: control <= {ADD, __, _1, __, __, __, _1, _1, _1, __}; // SUBI
    10'b00110_00: control <= {ADD, __, __, _1, __, __, _1, _1, _1, __}; // ADCI
    10'b00111_00: control <= {ADD, __, _1, _1, __, __, _1, _1, _1, __}; // SB CI
    10'b01000_00: control <= {AND, __, __, _x, __, __, _1, _1, __, __}; // ANDI
    10'b01001_00: control <= {OR, __, __, _x, __, __, _1, _1, __, __}; // ORI
    10'b01010_00: control <= {XOR, __, __, _x, __, __, _1, _1, __, __}; // XORI
    10'b01011_00: control <= {ADD, __, _1, __, __, __, _1, __, _1, __}; // CMPI
    10'b01100_00: control <= {AND, __, __, _x, __, __, _1, __, __, __}; // TSTI
    10'b01101_00: control <= {OR, _1, __, _x, __, __, _1, _1, __, __}; // LDI

    10'b01111_00: control <= {OR, _1, __, _x, __, __, __, _1, __, __}; // RORI
    10'b01111_01: control <= {OR, _1, __, _1, RORC, __, _1, _1, __, __}; // RORC
    10'b01111_10: control <= {OR, _1, __, _x, ASR, __, _1, _1, __, __}; // SHR
    10'b01111_11: control <= {OR, _1, __, _x, ASRA, __, _1, _1, __, __}; // SHRA

    // alu za ib xa rors imm wd wc wz pca
    10'b01111_101_00: control <= {OR, _1, _1, _x, __, __, __, _1, __, __}; // NOT
    10'b01111_101_01: control <= {ADD, _1, _1, __, __, __, __, _1, _1, __}; // NEG
    10'b01111_101_10: control <= {OR, _1, __, _x, __, __, __, _1, __, __}; // MOV
    10'b01111_110_0x: control <= {xx, _x, _x, _x, xx, _x, _1, __, __, __}; // RDPC/RDFL
    10'b01111_111_00: control <= {xx, _x, _x, _x, xx, _x, _1, __, __, __}; // LDPC/JMPx
    10'b01111_111_1x: control <= {OR, _1, __, _x, __, __, __, __, __, __}; // JIND/RETI
    10'b10000_00: control <= {ADD, __, __, __, __, __, _1, _1, __, __}; // LD
    10'b10001_00: control <= {ADD, __, __, __, __, __, _1, _1, __, __}; // LDB
    10'b10011_00: control <= {ADD, __, __, __, __, __, _1, _1, __, __}; // LDBS
    10'b10100_00: control <= {ADD, __, __, __, __, __, _1, __, __, __}; // ST
    10'b10101_00: control <= {ADD, __, __, __, __, __, _1, __, __, __}; // STB
    10'b1011x_00: control <= {ADD, __, __, __, __, __, _1, __, __, __}; // JR
    10'b1011x_01: control <= {ADD, __, __, __, __, __, _1, _1, __, __}; // JAL
    10'b11xxx_00: control <= {ADD, __, __, __, __, __, _1, __, __, __}; // Jcc

  default:
    control <= {xx, _x, _x, _x, xx, _x, __, __, __, __}; // ILEG/NOP
  endcase

```

```
// Writing to regs

assign dwr = opval & wd & (~jmp); // register bank write
assign wrcv = opval & wc; // C and V flags write
assign wrzn = opval & wz; // Z and N flags write

endmodule
//-----
```

