GUSY-VGA

Jesús Arias Alvarez

Dpto. de Electicidad y Electrónica. E.T.S.I. Telecomunicación.

Universidad de Valladolid

1 The GUSY computer

The GUSY computer is an implementation of the GUS16 CPU in an FPGA along with several peripherals, and most notably, a video controller. The block diagram of such computer is shown in figure 1. This computer can be synthesized in the SIMRETRO board which includes an ICE40HX4K FPGA, a 64K×16 SRAM memory, an 8MB SPI flash memory, and a VGA interface including three resistor-based digital to analog converters.

The video mode implemented is the standard VGA mode of 640×480 pixels, but, in order to fit the required video memory into the existing RAM the resolution was reduced to 512×400 pixels, with four bits per pixel.

The FPGA includes some RAM with a known initial content and in this design it was used as a boot memory, while the whole video memory is stored in the external SRAM of the board.

About 1200 logic cells of the FPGA are used in the implementation of the GUS16 CPU and the peripherals. The CPU itself takes about 750 logic cells, which is a remarkably low figure taking into account it is a 16-bit processor with an 8-register bank. This is less than a synthetic 6502 with around 850 logic cells or a Z80 with 2200 logic cells.

The description of the processor and peripherals follows.

1.1 CPU

The diagram of the GUS16 CPU is shown in figure 2. It has a Von-Neumman design and a two-stage pipeline with a fetch unit, which includes the program counter, the memory, and the instruction register, and an execution unit composed mainly by the register bank and the arithmetic-logic-unit. The execution of an instruction takes two clock cycles, one for fetching the op-code from the memory and another for the execution of the instruction, but as these units operate in parallel one instruction is finished on every cycle. There are exceptions, however. Load and store instructions have to access the memory during their execution phase and therefore it is not possible to read an op-code during that cycle. This results in the load of an invalid op-code in the IR register that has to be discarded, so, load and store instructions take two effective clock cycles to execute. The same happens during program jumps because the op-code loaded into IR is the one following the jump instead of that at the destination address, so, all taken jumps results also in two effective execution cycles. In summary: load, store, and jumps take two cycles while the rest of instructions are executed in just one cycle.

Conditional jumps depends on the value of a flag register that includes the usual four carry, zero, negative, and overflow flags. Also, the ALU can perform the adding, subtraction, AND, OR, and exclusive-OR of its two input operands depending on a few control lines. Its processing is complemented with a multiplexer that enables the shifting and/or rotation of data one bit to the right. No multiplication, division, nor multi-bit shift is possible in the GUS16 ALU.

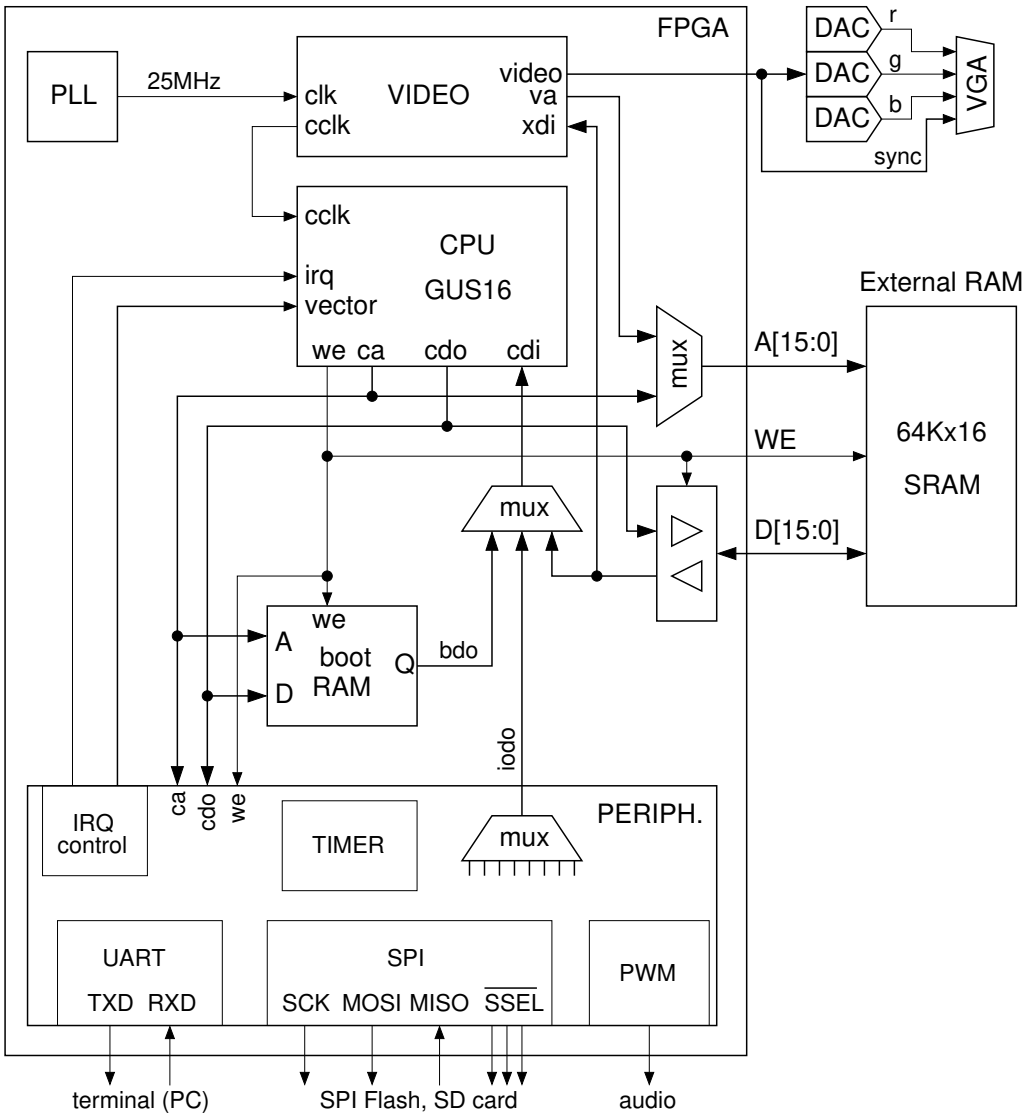

Figure 1: Block diagram of the GUSY computer.

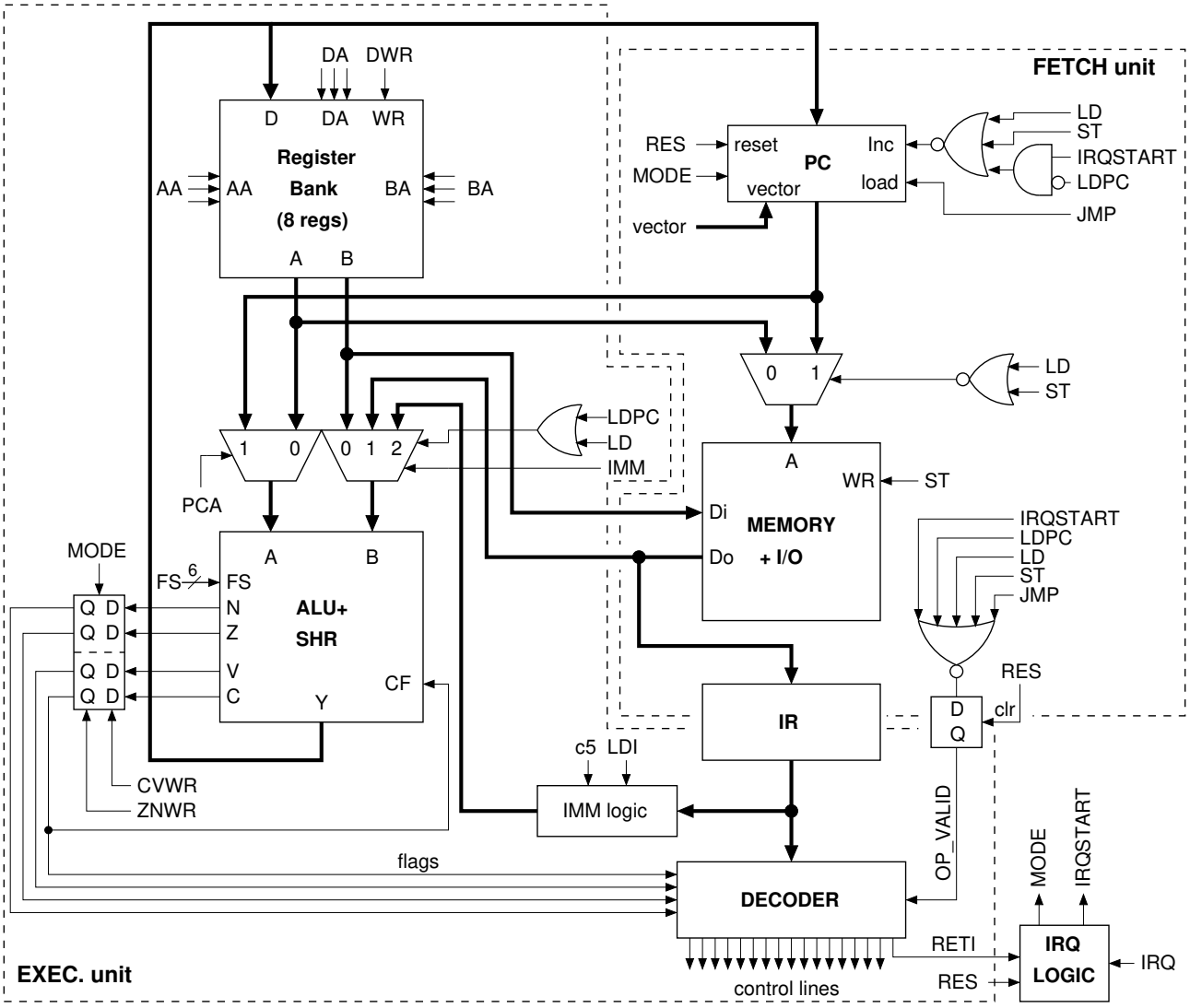

Figure 2: Block diagram of the GUS16 CPU

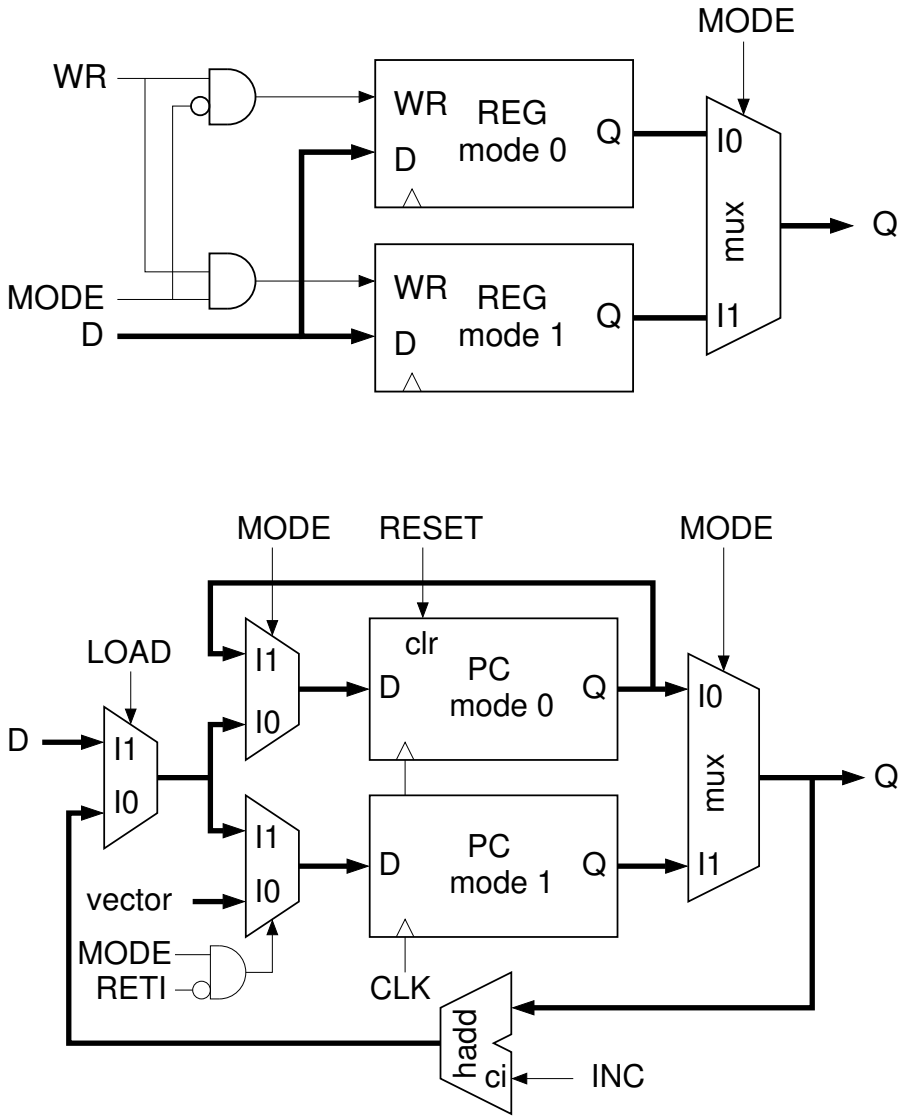

Figure 3: Block diagram of the dual mode registers: Flags and PC

The only specific purpose registers are the PC and the flags. There is no dedicated stack pointer in the GUS16 CPU. Yet, an stack can be implemented by software allowing the nesting of subroutine calls. This CPU also supports interrupts, and it was not an easy feat in a CPU without stack. The way the status of the main program is saved when an interrupt happens is by switching to different PC and flags registers during the execution of the interrupt routine.

The PC and flag registers include a MODE input to select between the normal and interrupt register to use, and internally there are two copies of these registers (see figure 3). In the PC case the program counter for interrupts (mode 1) remains precharged with the address of the interrupt routine (vector) until the MODE signal changes to high. There are just one incrementing logic (half-adder) for the two PC registers.

By having separate PC and flags for the normal operation mode and for the interrupt mode there is no need to spend time saving and restoring these registers. The limitation of this approach is the impossibility of the nesting of interrupts. Of course, the registers in the bank have to be saved and restored by program.

The change to the interrupt mode involves the setting of the MODE signal, but before doing this the current instruction has to be executed completely in order to avoid unexpected results. Therefore during just one cycle an IRQSTART signal is asserted. This signal forces the marking of the IR contents as invalid, just before the changing of the MODE signal (see figure 4) The same would happen at the end of the interrupt routine when the RETI instruction gets executed. But in this case the RETI instruction is in fact a taken jump and the IR register is also marked as not valid before returning to the normal mode of execution. The chronograph and the

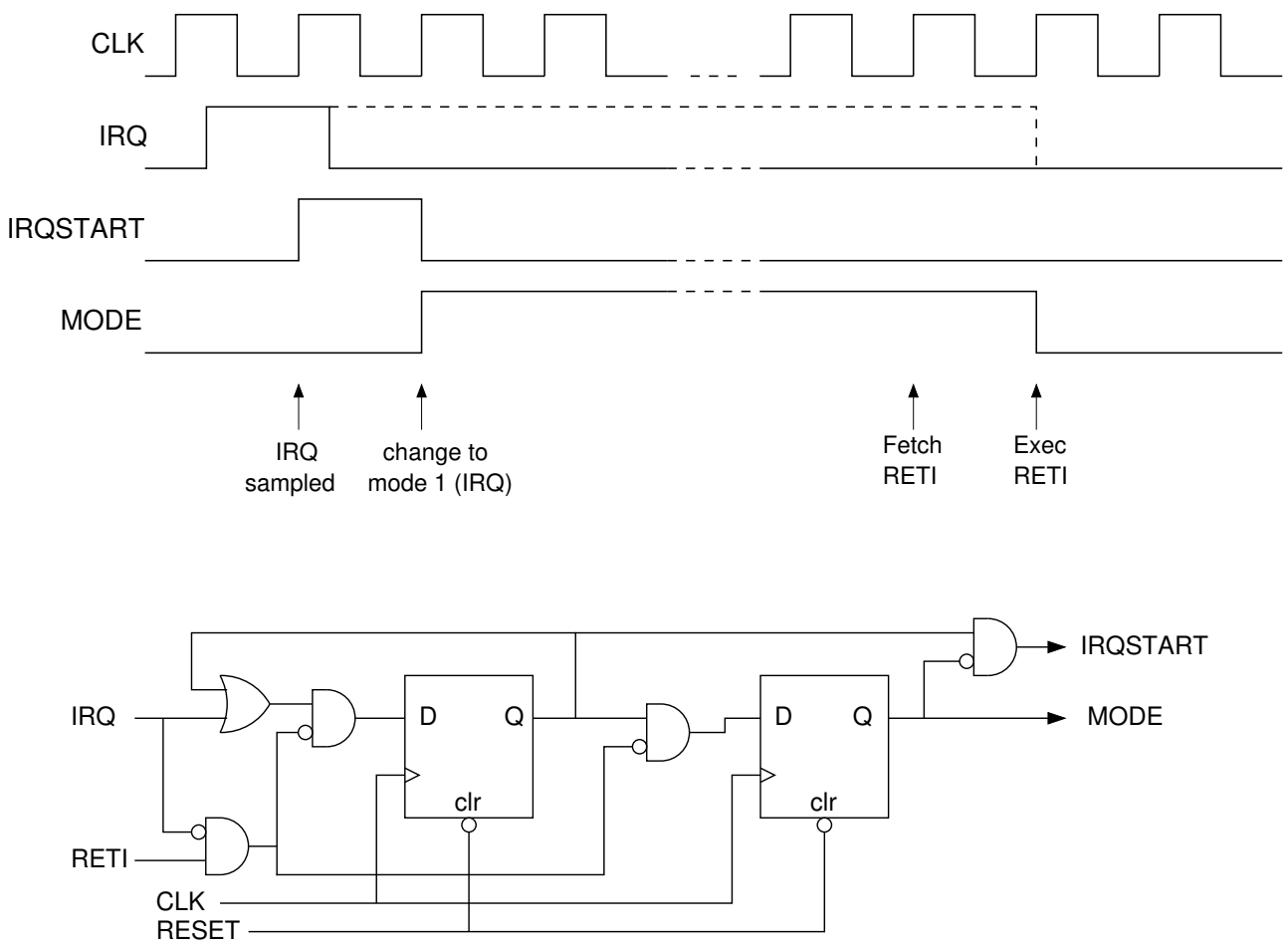

Figure 4: Chronograph and logic of interrupts in the GUS16 CPU

logic of interrupts are shown in figure 4.

An additional feature of the interrupt logic is the chaining of interrupts: If the IRQ signal remains active when the RETI instruction is executed the MODE signal remains high while a new interrupt vector gets loaded into the PC.

1.2 Instruction set

The instruction set of the GUS16 processor is summarized in figure 5, along with the coding and the flags modified for each instruction. It totals 38 different instructions that can be classified into the following categories:

- Data processing: ADD, ADC, SUB, SBC, CMP, AND, OR, XOR, TST, NOT, NEG
- Data processing with 4-bit immediate operand: ADDI, ADCI, SUBI, SBCI, CMPI, ANDI, ORI, XORI, TSTI, ADPC
- Shift and rotation to the right; SHR, SHRA, ROR
- Load and Store: LD, ST, LDPC
- Load immediate (8-bit data): LDI
- Jumps: JIND, RETI, JR, JZ, JNZ, JC, JNC, JMI, JPL, JV

The way an stack pointer is implemented involves using two instructions for pushing and poping. Let assume R7 is the stack pointer:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	mnemonic	flags				operation
0	0	0	0	0	RD			0	RA			–	RB			ADD	C	V	N	Z	RD= RA+RB
0	0	0	0	0	RD			1	RA			data				ADDI	C	V	N	Z	RD= RA+data
0	0	0	0	1	RD			0	RA			–	RB			ADC	C	V	N	Z	RD=RA+RB+Cflag
0	0	0	0	1	RD			1	RA			data				ADCI	C	V	N	Z	RD=RA+dato+Cflag
0	0	0	1	0	RD			0	RA			–	RB			SUB	C	V	N	Z	RD=RA−RB
0	0	0	1	0	RD			1	RA			data				SUBI	C	V	N	Z	RD=RA−data
0	0	0	1	1	RD			0	RA			–	RB			SBC	C	V	N	Z	RD=RA−RB−(~Cflag)
0	0	0	1	1	RD			1	RA			dato				SBCI	C	V	N	Z	RD=RA−dato−(~Cflag)
0	0	1	0	0	–	–	–	0	RA			–	RB			CMP	C	V	N	Z	RA−RB
0	0	1	0	0	–	–	–	1	RA			data				CMPI	C	V	N	Z	RA−data
0	0	1	0	1	RD			0	RA			–	RB			AND	–	–	N	Z	RD=RA&RB
0	0	1	0	1	RD			1	RA			data				ANDI	–	–	N	Z	RD=RA&data
0	0	1	1	0	–	–	–	0	RA			–	RB			TST	–	–	N	Z	RA&RB
0	0	1	1	0	–	–	–	1	RA			data				TSTI	–	–	N	Z	RA&data
0	0	1	1	1	RD			0	RA			–	RB			OR	–	–	N	Z	RD=RA|RB
0	0	1	1	1	RD			1	RA			data				ORI	–	–	N	Z	RD=RA|data
0	1	0	0	0	RD			0	RA			–	RB			XOR	–	–	N	Z	RD=RA^RB
0	1	0	0	0	RD			1	RA			data				XORI	–	–	N	Z	RD=RA^data
0	1	0	0	1	RD			0	–	–	–	–	RB			NOT	–	–	N	Z	RD=~Rb
0	1	0	0	1	RD			1	–	–	–	–	RB			NEG	–	–	N	Z	RD=−RB
0	1	0	1	0	RD			0	–	–	–	–	RB			SHR	C	?	N	Z	RD=RB/2, Cflag=RB.0
0	1	0	1	0	RD			1	–	–	–	–	RB			SHRA	C	?	N	Z	RD=RB/2, Cflag=RB.0 (signed)
0	1	0	1	1	RD			0	–	–	–	–	RB			ROR	C	?	N	Z	RD=(RB>>1)|(Cflag<<15), Cflag=RB.0
0	1	0	1	1				1								ILEG					
0	1	1	0	0	RD			0	RA			–	–	–	–	LD	–	–	N	Z	RD=Mem[RA]
0	1	1	0	0	–	–	–	1	RA			–	RB			ST	–	–	–	–	Mem[RA]=RB
0	1	1	0	1	RD			0	–	–	–	data				ADPC	–	–	–	–	RD=PC+data
0	1	1	0	1	–	–	–	1	–	–	–	0	RB			JIND	–	–	–	–	PC=RB
0	1	1	0	1	–	–	–	1	–	–	–	1	–	–	–	RETI	–	–	–	–	Return from Interrupt
0	1	1	1	0	RD			–	–	–	–	–	–	–	–	LDPC	–	–	–	–	RD=Mem[PC++]
0	1	1	1	1	RD			dato								LDI	–	–	–	–	RD=dato (8 bits)
1	0	0	0	signed offset												JZ	–	–	–	–	jump if Zflag=1
1	0	0	1	signed offset												JNZ	–	–	–	–	jump if Zflag=0
1	0	1	0	signed offset												JC	–	–	–	–	jump if Cflag=1
1	0	1	1	signed offset												JNC	–	–	–	–	jump if Cflag=0
1	1	0	0	signed offset												JMI	–	–	–	–	jump if Nflag=1 (negative)
1	1	0	1	signed offset												JPL	–	–	–	–	jump if Nflag=0 (positive)
1	1	1	0	signed offset												JV	–	–	–	–	jump if Vflag=1 (overflow)
1	1	1	1	signed offset												JR	–	–	–	–	unconditional jump

Figure 5: Instruction set of the GUS16 CPU

```
; Push R0 (full-descending stack)
SUBI R7,R7,1
ST (R7),R0
; Pop R0 (full-descending stack)
LD R0,(R7)
ADDI R7,R7,1
```

In order to call a leaf subroutine (a routine with no nested subroutine calls) we can use the following code:

```
ADPC R6,1 ; R6 contains the return address
JR routine
```

The PC points one instruction in advance of the one being executed. Therefore R6 gets loaded with the address of the instruction that follows JR that is precisely where the subroutine has to return.

And for the returning from a subroutine just execute:

```
JIND R6
```

Of course, the value of R6 has to be preserved during the execution of the subroutine, and this includes saving it on the stack before calling other subroutines and restoring it after these calls.

There are no instructions for shifting data to the left, but these are not needed because the same result can be obtained by adding the source register to itself. Also, the carry flag can be included in a rotation to the left:

```
ADD R1,R0,R0 ; R1=SHR(R0)
ADC R1,R0,R0 ; R1=ROL(R0)
```

There are no MOV instructions for the data transfer between registers. If needed, use an arithmetic or logic instruction. For instance:

```
OR   R1,R0,R0
ORI  R1,R0,0
ADDI R1,R0,0
```

Any of these instructions moves the contents of R0 to R1, but they also change the flags. The first two instructions only changes the Zero and Negative flags depending on the data, while the last instruction also clears the carry flag.

1.3 Memory map

The GUS16 CPU can address up to 64Kword. In these address space the boot RAM, the external RAM, and the I/O registers have to be mapped. In figure 6 the relevant areas of the memory map are shown. The first 4K are mapped to the boot RAM, with the exception of 32 words reserved for I/O registers. The remaining 60K address range selects the external RAM. In this area 50Kwords are reserved for video memory, with each word containing 4 pixels, and 10Kwords are free. Notice that the external RAM is contended between the CPU and the video controller. If the video controller is reading a word and the CPU tries to access the external memory during the same cycle the CPU clock is forced high and a wait state is inserted (the video controller always win the contention). The video controller reads a word every 4 cycles during the visible part of the image. Also notice that there is no contention if the CPU is accessing the boot RAM or I/O registers. The effective CPU speed when executing code from the external RAM is thus an 88% of its nominal value or 22MHz.

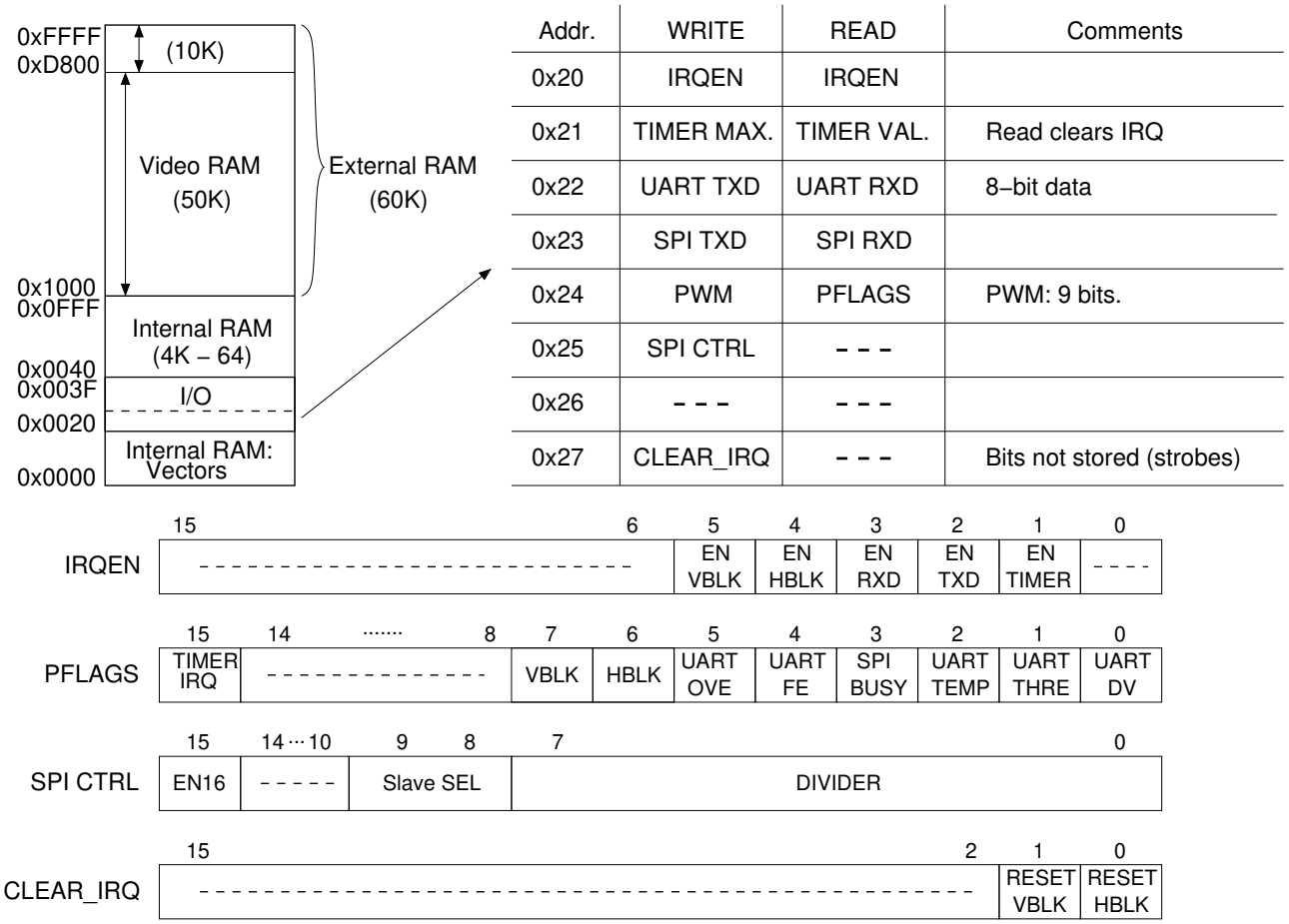

Figure 6: Memory map, including the I/O block, and bit fields of I/O registers

1.4 Peripherals

In figure 6 the registers of the included peripherals are also shown. These peripherals are:

1.4.1 UART

The UART is a simple one with a fixed speed of 115200 bps and a fixed data format of 8 bits, 1 stop bit and no parity. There are two data registers for receiver and transmitter and several flags. The data registers have 8 bits and therefore the bits 8 to 15 are read as zero and ignored on writes. Both the transmitter and the receiver include a holding register (1-byte FIFO) in addition to its corresponding shifting register. The flags can be read in the PFLAGS register, where all the peripheral flags are located, and are:

- DV: Data Valid. Set to one when a character is received and cleared by reading the data register. This flag can request an interrupt when set.
- THRE: Transmitter holding register empty. Set to one when a character can be queued for transmission and cleared when both the holding register and the shift register are filled with data. This flag can request an interrupt when set.
- TEMP: Transmitter empty. Set to one when transmission is completed (holding register and shift register empty).
- FE: Frame error. Set to one when a character is received with its stop bit as zero.
- OVE: Overrun. Set to one if a character is received with DV already active.

1.4.2 SPI controller

The SPI controller allows the exchange of 8-bit or 16-bit data words with selectable clock rates. It includes a data write register, a data read register, a control register, and a single BUSY flag in the PFLAGS register. A write to the data register starts the transfer and sets the BUSY flag. This flag is cleared when the data transfer is completed. In the control register we have three bit fields:

- EN16 (bit 15): Enable 16-bit transfers when one and 8-bit transfers if zero. For 8-bit transfers the bits 8 to 15 are ignored on writes and they are read as zero.
- Slave SEL (bits 8 and 9): These two bits are decoded and they select up to three possible slave devices for the transfer (only two currently implemented):

Slave SEL	Device selected
00	none
01	Flash SPI
10	SD card
11	not implemented

- DIVIDER (bits 0-7): Select the frequency of the SCK signal according to the following formula:

$$f_{SCK} = \frac{f_{CLK}}{2 \times (DIVIDER + 1)} = \frac{12.5MHz}{(DIVIDER + 1)}$$

1.4.3 Timer

A simple 16-bit timer allows the generation of periodic interrupts. It includes two registers, one of them is the actual counter while the other holds the maximum count. When the counter reaches the value of maximum count on the next cycle it gets loaded with zero and an interrupt flag is set. This flag can be read in the PFLAGS register (bit 15).

A write to the timer register loads the maximum count and also resets the counter while a read returns the current count and clears the interrupt flag.

1.4.4 PWM

The PWM peripheral is included for the generation of sampled audio. Instead of using the 16-bit timer as its base counter the PWM relies on the horizontal counter of the video generator. Therefore a PWM cycle is 32μs long and it is in sync with the VGA video. The PWM pulse starts as high during the horizontal blank interval and goes low when the current pixel reaches the value of the PWM register. This register is 9-bit wide and its value range from 0 (36% duty) to 511 (99.8% duty). Applications must write the PWM value during the horizontal blanking interval to avoid the generation of spurious glitches.

1.5 Interrupts

There are 5 interrupt sources in the design and each one can be masked individually by writing a zero in its corresponding bit in the IRQEN register. These are:

- The Timer interrupt, enabled by bit 1 of IRQEN
- The UART THRE interrupt, enabled by bit 2 of IRQEN
- The UART DV interrupt, enabled by bit 3 of IRQEN

- The Horizontal blanking interrupt, enabled by bit 4 of IRQEN
- The Vertical blanking interrupt, enabled by bit 5 of IRQEN

The interrupts are arranged in order of increasing priority, with the vertical blanking interrupt being the one with the higher priority. If more than one interrupt is active the one with the higher priority will have its vector written into the PC register of the CPU when the mode switches to interrupt. The corresponding vectors are listed in the following table:

IRQ #	Source	Vector address	Comments
—	Reset	0x0	not an IRQ (forces mode 0)
0	unused	0x4	Lowest priority
1	Timer	0x8	
2	UART THRE	0xC	
3	UART DV	0x10	
4	HBLK	0x14	
5	VBLK	0x18	
6	unused	0x1C	Highest priority

The HBLK and VBLK interrupts are requested at the beginning of the corresponding video blanking interval, but these requests are not cleared until a write to the CLEAR_IRQ register is done. In this pseudo-register if bit 0 is set the HBLK interrupt is cleared and if bit 1 is set the VBLK interrupt is cleared. The data written to CLEAR_IRQ is not stored.

1.6 Video generation

In the standard VGA mode a line is 32μs long, and by using a 25MHz clock, this results in a total of 800 pixels. Only 512 pixels are actually visible (in the VGA mode there are 640 visible pixels), the rest of the line is a blanking interval. During this blanking interval the HSYNC pulse is generated.

The same happens in the vertical dimension: There are a total of 525 lines, with 400 lines being displayed (In the VGA mode there are 480 visible lines).

The resolution was reduced in order to fit the video memory into the available RAM, and also to have an horizontal resolution that is a power of two to avoid multiplications in the software. This resolution results in 200Kpixels, and with a color depth of 4 bit per pixel (16 simultaneous colors in the screen), each memory word stores four pixels. Thus, 50Kwords are needed for the framebuffer. Its base address is 0x1000, that corresponds to the beginning of the external memory in the memory map.

In each word, the most significant bits corresponds to the pixel displayed on the left of the screen, and the color format of each pixel is IRGB, meaning there is a bit for each color component plus an intensity bit that adds some gray to the color when one. The resulting color palette is:

Pixel value (IRGB)	Color	% R	% G	% B
0000	Black	0	0	0
0001	Blue	0	0	67
0010	Green	0	67	0
0011	Cyan	0	67	67
0100	Red	67	0	0
0101	Magenta	67	0	67
0110	Yellow	67	67	0
0111	Grey	67	67	67
1000	Dark Grey	33	33	33
1001	Light Blue	33	33	100
1010	Light Green	33	100	33
1011	Light Cyan	33	100	100
1100	Light Red	100	33	33
1101	Light Magenta	100	33	100
1110	Light Yellow	100	100	33
1111	White	100	100	100

In order to obtain the listed color weights by using 4-bit DACs for the R, G, and B components, the bits driving these DACs are:

- R[3:0]=RIRI
- G[3:0]=GIGI
- B[3:0]=BIBI

With these bit assignments, each component can have a value of 0, 5, 10, or 15 (0%, 33%, 67%, or 100%).