The GUS-16 V6 CPU

Jesús Arias

1 The GUS-16 evolution

This CPU core of my own design has changed many times since its first conception, some times a little, others drastically. It started as an design example for students, and that implied a design as simple as possible, but with also a practical use in mind, so, it couldn't be too simple neither. As a performance minimum I chose to have some way to call and return from a subroutine. The core started as a 16-bit processor with a register bank with 8 registers and two output busses to feed an ALU, a 16-bit program counter, and a 4-bit flag register. These basic features were maintained through all the versions of the core, that are:

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

The original core was a Harvard design with separate program and data memories that executed all instructions in a single clock cycle and was named simply "CPU_ONE_CYCLE". That version was quickly abandoned in favor of a Von Neumman implementation due to the problems it had when dealing with constants (take for instance how to print a character string "Hello World"). Later versions had a single memory for instructions and data and this was no longer a big problem, yet, the loading of constants was always a concern. They also have a two-stage pipeline that allows an effectice one-cycle execution time for instructions other than load, store, and jumps, that required two-cycles.

Version V3 added support for interrupts. This was accomplished by adding a second PC and flags registers that are switched in when an interrupt happens, leaving the main PC and flags with their last values unmodiffied until the new instruction RETI is executed. This was the first version being written in Verilog and synthesized in a FPGA.

Early versions included the LDH instruction for loading the high byte of 16-bit constants into an hidden register. That instruction was removed in the version V4 and replaced by LDPC instead. This new instruction uses the PC as a pointer to load the 16-bit word that follows the current instruction into an Rx register. This has no advantages from the software point of view, but simplifies the hardware design that no longer requires an 8-bit hidden register to store the high bytes of constants.

Version V5 added 4-bit positive literal displacements to the memory address during load and store instructions. This "small" change required a substantial rearrangement of the processor internals in order to use the ALU for the address computation. On the other hand the inclusion of displacements was a quite nice feature for software development that saved many instructions, for instance when dealing with stacks.

The lastest V6 version includes big changes in the instruction set, while the processing hardware is almost the same with the exception of a new barrel shifter that enables single-cycle multi-bit rotations.

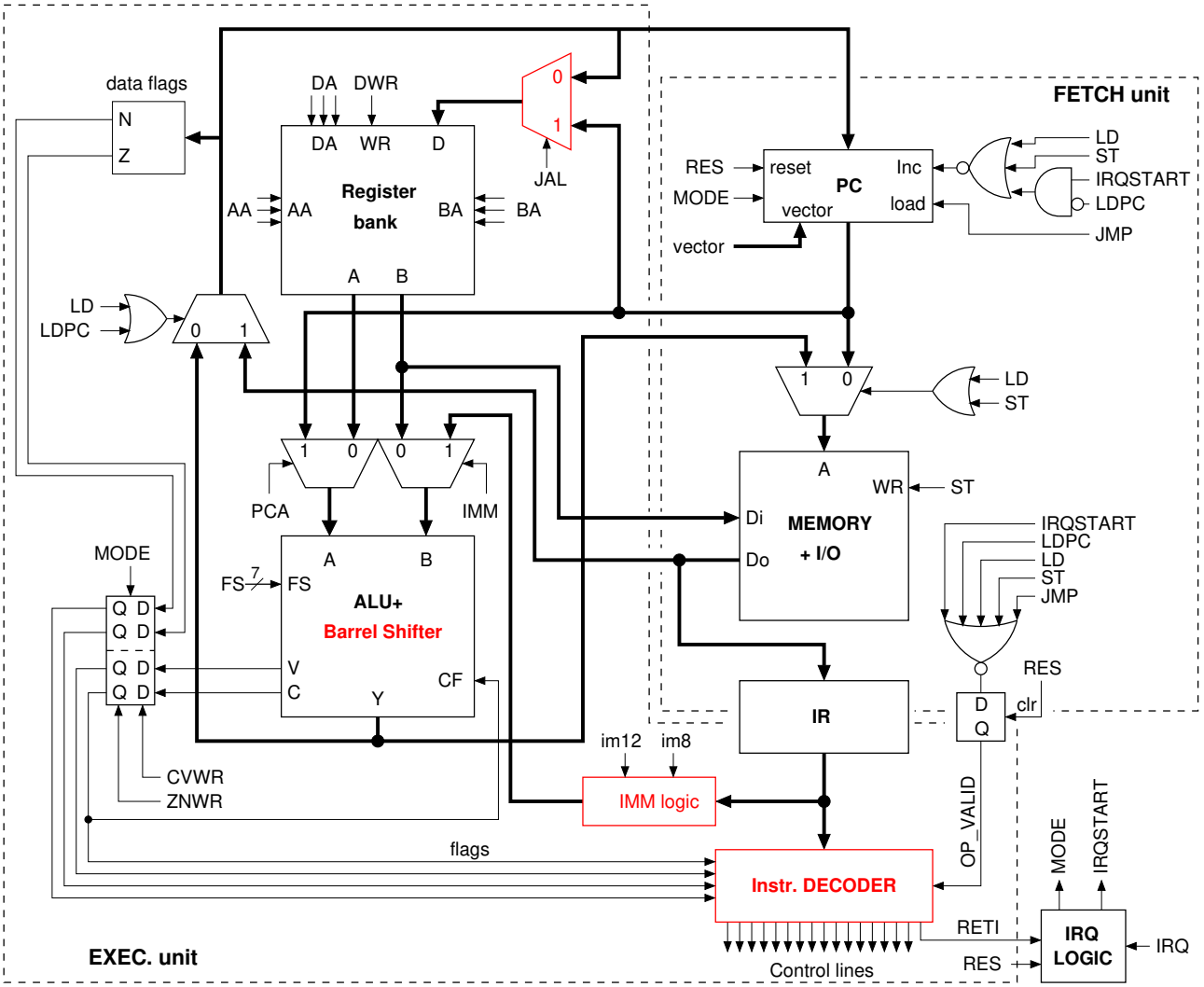

Figure 1: Block diagram of the GUS16-V6 processor. New hardware is marked in red

2 GUS16-V6 features

The new V6 processor has the diagram shown in figure 1. This includes a new multiplexer for the register bank input in order to route the content of the PC register into R6 during the execution cycle of the JAL instruction. Other changes are present in the ALU, the immediate operand logic, and of course in the decoding.

The ALU and immediate operand logic blocks are shown in more detail in figure 2, where as we can see, the output of the ALU is followed by a 16-bit rotator. The rotator is built using 4 stacked multiplexers: the first selects between the input data or the same data rotated 8 bits, the second between the input data or the same data rotated 4 bits, and so on. This rotator is used by the RORI instruction, but also by single bit shifts. Because of this its control lines can be forced to one (for shifts) or zero (for any instruction other than shifts or RORI), and its LSB can be selected to be the LSB of the ALU, 'f0', for non-shift instructions (also for RORI), zero for the shift-right, SHR, instruction, the sign bit, 'f15', for the shift-right arithmetic, SHRA, instruction, of the carry flag for the RORC instruction.

The immediate operand logic is quite simple: the lower 5 bits of the instruction register are always present at the output while the higher bits are forced to zero for positive literals, starting at bit #5 for LD/ST, or bit #8 for other immediate operands. In the case of jumps the displacements are 12-bit signed literals, and therefore in these cases the bit #11 is copied into the 4 upper bits (sign extension).

The instruction set of the V6 processor is summarized in figure 3. Its new features include:

- Jump and link instruction, JAL. This instruction adds a signed 12-bit displacement to PC (like the in-

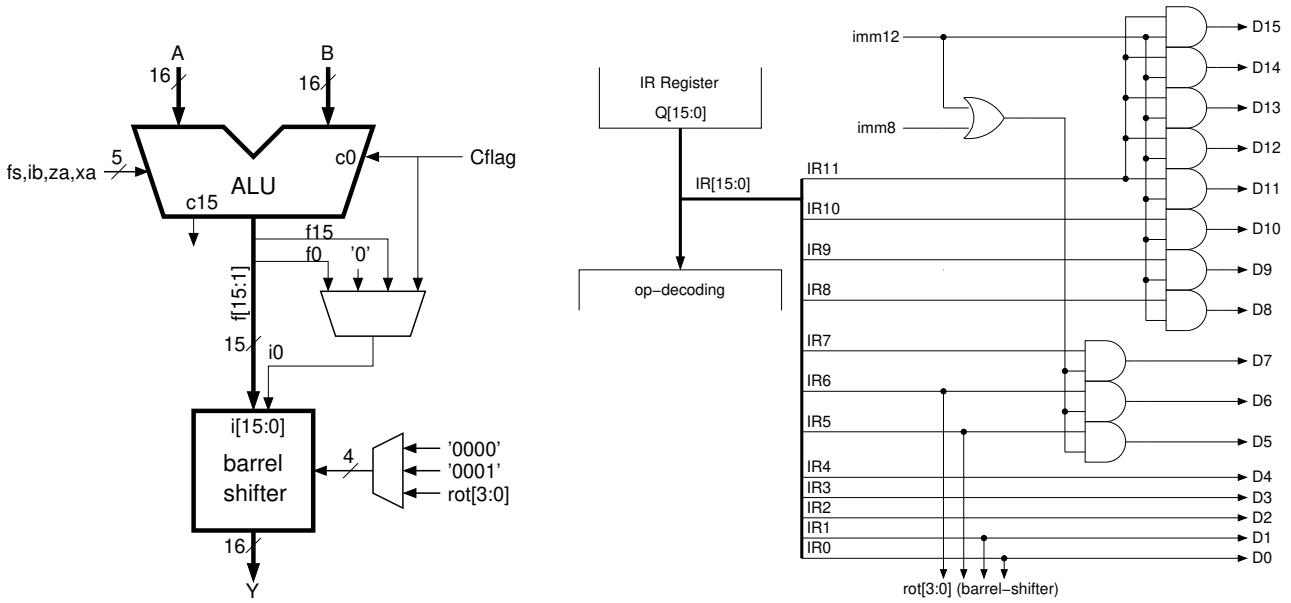

Figure 2: Detail of the ALU and the immediate operand logic

struction JR), and also saves the current PC value to R6. This reduces the call to a subroutine to a single instruction instead of the two instructions used before:

```
V5:                                    V6:
call:    ADPC   R6,1 ; R6 = PC+1       call:    JAL routine  ; R6 = PC
         JR     routine                rethere: ...
rethere: ...
return:  JIND R6                       return:  JIND R6
```

- Multibit rotation, RORI Rd, Rs, ulit4. This new instruction rotates the contents of register Rs "ulit4" positions to the right and leaves the result in register Rd. Notice that a rotation to the left is simply a (16-nbit) rotations to the right, thus, this instruction can also performs as "ROL". The carry flag isn't included in the rotation. The old ROR instruction that included the carry in a single bit rotation is maintained, but it is now called RORC. The new RORI instruction finds a good use as a byte swap (RORI Rd,Rs,8), but it also simplifies arithmetic. Take for instance a multiplication by 40:

```
V5:                                V6:
    ADD R0,R0,R0                       RORI R1,R0,11 ; R1=R0*32
    ADD R0,R0,R0                       RORI R0,R0,13 ; R0=R0*8
    ADD R0,R0,R0 ; R0=R0*8             ADD  R0,R0,R1 ; R0=R0*40
    ADD R1,R0,R0
    ADD R1,R1,R1 ; R1=R0*32
    ADD R0,R0,R1 ; R0=R0*40
```

- Now, for other instructions with immediate operands the size of the literal is 8 bits instead of 4. This was achieved as the expense of fewer instructions (CMP, TST, and TSTI were eliminated) and also by having the same register as the source and destination. (In fact, the source and destination registers were the same in much of the V5 code sources)

```
V5:                                V6:
    ADDI R2,R1,4 ; R2 = R1+4           ADDI R1,127 ; R1 = R1+127
    CMPI R0,15                         CMPI R0,255
```

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			RA			RB			0	0	ADD	C	V	N	Z	RD= RA + RB
0	0	0	0	0	RD			RA			RB			0	1	SUB	C	V	N	Z	RD= RA – RB
0	0	0	0	0	RD			RA			RB			1	0	ADC	C	V	N	Z	RD= RA + RB+Cflag
0	0	0	0	0	RD			RA			RB			1	1	SBC	C	V	N	Z	RD= RA– RB –~cflag
0	0	0	0	1	RD			RA			RB			0	0	AND	–	–	N	Z	RD= RA & RB
0	0	0	0	1	RD			RA			RB			0	1	OR	–	–	N	Z	RD= RA | RB
0	0	0	0	1	RD			RA			RB			1	0	XOR	–	–	N	Z	RD= RA ^ RB
0	0	0	0	1	RD			RA			RB			1	1	BIC	–	–	N	Z	RD= RA & (~RB)
0	0	0	1	0	RD			ulit8								ADDI	C	V	N	Z	RD= RD + ulit8
0	0	0	1	1	RD			ulit8								SUBI	C	V	N	Z	RD= RD – ulit8
0	0	1	0	0	RD			ulit8								ADCI	C	V	N	Z	RD= RD + ulit8 +Cflag
0	0	1	0	1	RD			ulit8								SBCI	C	V	N	Z	RD= RD – ulit8 –~Cflag
0	0	1	1	0	RD			ulit8								ANDI	–	–	N	Z	RD= RD & ulit8
0	0	1	1	1	RD			ulit8								ORI	–	–	N	Z	RD= RD | ulit8
0	1	0	0	0	RD			ulit8								XORI	–	–	N	Z	RD= RD ^ ulit8
0	1	0	0	1	RD			ulit8								CMPI	C	V	N	Z	RD – ulit8
0	1	0	1	0	RD			ulit8								LDI	–	–	–	–	RD= ulit8
0	1	0	1	1	RD			0	ulit4h		RB			ulit4l		RORI	–	–	N	Z	RD = (RB>>uli4) | (RB<<16–ulit4)
0	1	0	1	1	RD			1	0	0	RB			0	0	RORC	C	?	N	Z	RD = {Cflag,RB>>1}, Cflag=RB0
0	1	0	1	1	RD			1	0	0	RB			0	1	SHR	C	?	N	Z	RD = RB>>1, Cflag=RB0
0	1	0	1	1	RD			1	0	0	RB			1	0	SHRA	C	?	N	Z	RD = RB>>1 (signed) , Cflag=RB0
0	1	0	1	1	RD			1	0	1	RB			0	0	NOT	–	–	N	Z	RD = ~RB
0	1	0	1	1	RD			1	0	1	RB			0	1	NEG	C	V	N	Z	RD = –RB
0	1	0	1	1	RD			1	1	1	–	–	–	0	0	LDPC	–	–	–	–	RD = Mem(PC++)
0	1	0	1	1	–	–	–	1	1	1	RB			1	0	JIND	–	–	–	–	PC = RB
0	1	0	1	1	–	–	–	1	1	1	–	–	–	1	1	RETI	–	–	–	–	return from interrupt
0	1	1	0	0	RD			RA			udisp5					LD	–	–	N	Z	RD= Mem(RA+udisp5)
0	1	1	0	1	RB			RA			udisp5					ST	–	–	–	–	Mem(RA+udisp5)=RB
0	1	1	1	sdisp12												JAL	–	–	–	–	PC = PC+sdisp12, Rlink=PC
1	0	0	0	sdisp12												JZ	–	–	–	–	PC = PC+sdisp12 if Zflag
1	0	0	1	sdisp12												JNZ	–	–	–	–	PC = PC+sdisp12 if ~Zflag
1	0	1	1	sdisp12												JC	–	–	–	–	PC = PC+sdisp12 if Cflag
1	0	1	1	sdisp12												JNC	–	–	–	–	PC = PC+sdisp12 if ~Cflag
1	1	0	0	sdisp12												JMI	–	–	–	–	PC = PC+sdisp12 if Nflag
1	1	0	1	sdisp12												JPL	–	–	–	–	PC = PC+sdisp12 if ~Nflag
1	1	1	0	sdisp12												JV	–	–	–	–	PC = PC+sdisp12 if Vflag
1	1	1	1	sdisp12												JR	–	–	–	–	PC = PC+sdisp12

Figure 3: GUS16-V6 Instruction Set

- The load, LD, and store, ST, instructions now have a 5 bit displacement. This is one bit more than in the V5 case and allows the addressing of words up to +31 positions up in the memory pointed by the base register. As in the V5 case, the displacement is always positive.
- A new bit clear, BIC, instruction is included, mainly because I had a free op-code available for it. But it lacks an immediate variant that would have been more useful. This instruction is thus a good candidate for removal. In fact I got a ROR Rd,Rb,Ra instruction in mind that could end using the same op-code instead (maybe for a V7 variant...).

3 V6 comparisons

The V5 instruction set is included in figure 4 for comparison (sorry for the Spanish version. I hope no Rosetta Stone is needed for its reading ;). The V5 lacks the JAL and RORI instructions but on the other hand it includes the useful CMP, TST, and TSTI instructions. All the immediate operands are 4-bit wide, with the exception of LDI (8-bits) and jumps (12-bits with sign).

The V5 processor lacks the barrel shifter and consequently it requires less logic for being implemented. A synthesis for a Lattice ICE40HX FPGA gave the following results:

Core	Logic Cells	max. Frequency (MHz)	Synth. time[1] (seconds)
GUS16-V5	673	56.49	3.98
GUS16-V6	760	40.56	4.07
6502 (Arlet Ottens)	789	54.15	5.25
Z80 (Guy Hutchison)	2247	43.24	25.46

The new V6 core requires 87 logic cells more than the V5. This is about a 13% more space. Notice the barrel shifter itself needs 64 logic cells, one per 2-to-1 multiplexer. The table also shows the result of the synthesis of two classic cores: a 6502 and a Z80. As we can see, the GUS16-V6 have almost the same size as an 8-bit 6502, while it requires only 1/3 of the space of an 8-bit Z80. The maximum clock frequency estimate is also lower for the V6 core. This can be explained by the additional delay introduced by the barrel shifter that is connected in series with the ALU. They could have been connected in parallel, but in that case another multiplexer would be needed in order to select the data from the ALU of the barrel shifter, and the maximum clock frequency is still quite high, so I chose the series connection in order to avoid the multiplexer.

3.1 Case study: the Floppyton-II firmware

The Floppyton-II is an embedded computer intended for the emulation of Apple-II floppy disks that is synthesized along the Apple-II replica, allowing to read and write floppy disk images stored in an SD card. The main components of this computer are:

- CPU: Z80 (original version), GUS16-V5, or GUS16-V6
- Memory: 12 Kbytes (Z80) or 6 Kwords (GUS16)
- OSD, text mode video controller (32 x 24 characters)
- PS2 keyboard input
- DMA for input and output floppy bit streams
- Fast SPI controller

[1] Synthesized on Intel I7 laptop

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	mnemónico	flags				operación
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			dato				ADDI	C	V	N	Z	RD= RA+dato
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			dato				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			dato				SUBI	C	V	N	Z	RD=RA−dato
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			dato				CMPI	C	V	N	Z	RA−dato
0	0	1	0	1	RD			0	RA			–	RB			AND	–	–	N	Z	RD=RA&RB
0	0	1	0	1	RD			1	RA			dato				ANDI	–	–	N	Z	RD=RA&dato
0	0	1	1	0	–	–	–	0	RA			–	RB			TST	–	–	N	Z	RA&RB
0	0	1	1	0	–	–	–	1	RA			dato				TSTI	–	–	N	Z	RA&dato
0	0	1	1	1	RD			0	RA			–	RB			OR	–	–	N	Z	RD=RA|RB
0	0	1	1	1	RD			1	RA			dato				ORI	–	–	N	Z	RD=RA|dato
0	1	0	0	0	RD			0	RA			–	RB			XOR	–	–	N	Z	RD=RA^RB
0	1	0	0	0	RD			1	RA			dato				XORI	–	–	N	Z	RD=RA^dato
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 (con signo)
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			d[3:0]				LD	–	–	N	Z	RD=Mem[RA+d]
0	1	1	0	0	d[2:0]			1	RA			d[3]	RB			ST	–	–	–	–	Mem[RA+d]=RB
0	1	1	0	1	RD			0	–	–	–	dato				ADPC	–	–	–	–	RD=PC+dato
0	1	1	0	1	–	–	–	1	–	–	–	0	RB			JIND	–	–	–	–	PC=RB
0	1	1	0	1	–	–	–	1	–	–	–	1	–	–	–	RETI	–	–	–	–	Retorna de Interrupcion
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	desplazamiento con signo												JZ	–	–	–	–	salto si Zflag=1
1	0	0	1	desplazamiento con signo												JNZ	–	–	–	–	salto si Zflag=0
1	0	1	0	desplazamiento con signo												JC	–	–	–	–	salto si Cflag=1
1	0	1	1	desplazamiento con signo												JNC	–	–	–	–	salto si Cflag=0
1	1	0	0	desplazamiento con signo												JMI	–	–	–	–	salto si Nflag=1 (negativo)
1	1	0	1	desplazamiento con signo												JPL	–	–	–	–	salto si Nflag=0 (positivo)
1	1	1	0	desplazamiento con signo												JV	–	–	–	–	salto si Vflag=1 (overflow)
1	1	1	1	desplazamiento con signo												JR	–	–	–	–	salto incondicional

Figure 4: GUS16-V5 Instruction Set

- Interrupts for track change detection.

The Floppyton firmware was written in assembler, first for a Z80, and then ported to the GUS16 cores. It includes about 2000 lines of source code and deals with SD card initialization, reading, and writing, FAT32 navigation, interactive file selection, and floppy emulation. This is a quite complex firmware and we have now 3 versions to compare. These are the numbers:

	Z80	GUS16 V5	GUS16 V6
Total memory[2] (bytes)	3522	3708	3406
Code size[3] (bytes)	2313	2548	2246
Code size vs Z80 size	100%	+10%	-3%
Code size vs GUS16 V5 size	-9%	100%	-12%

As we can see in the preceding table, the GUS16 V6 has the smaller code size, about a 12% less than the previous GUS16 V5, and even less than the original Z80 code. Both GUS16 processors were also much faster than the Z80: All versions ran with a 25MHz clock, but the Z80 takes many clock cycles per instruction. And the GUS16 core sizes are only 1/3 of the Z80's.

And now, some instruction usage statistics for the GUS16 V6:

ADD	18	ADDI	56	LDI	169	JIND	24	JNC	4
SUB	16	SUBI	73	RORI	23	RETI	1	JMI	1
ADC	6	ADCI	5	RORC	1	LD	172	JPL	13
SBC	0	SBCI	3	SHR	5	ST	169	JV	0
AND	7	ANDI	20	SHRA	0	JAL	90	JR	35
OR	46	ORI	6	NOT	0	JZ	40		
XOR	3	XORI	0	NEG	0	JNZ	53		
BIC	0	CMPI	33	LDPC	51	JC	4		

Of course, other applications can have different statistics, but as a general rule we can remark the 3-register instructions have less usage than their immediate operand version, with the exception being OR. But OR was used quite often to move data between registers (MOV Rd,Rs = OR Rd,Rs,Rs), at that explains its more than normal use. SHRA and JV aren't used in the firmware, but that's because the Floppyton code only deals with unsigned integer variables. On the other hand, BIC, NOT, and NEG are good candidates for removal, while XORI is a less clear one.

[2] 16-bit words accounted as 2 bytes

[3] Excluding character table, strings, and reserved areas