





Article

An Educational RISC-V-Based 16-Bit Processor

Jecel Mattos de Assumpção, Jr. ¹, Oswaldo Hideo Ando, Jr. ², Hugo Puertas de Araújo ¹ and Mario Gazziro ^{1,*}

¹ Information Engineering Group, Department of Engineering and Social Sciences (CECS), Federal University of ABC (UFABC), Av. dos Estados, 5001, Santo André 09210-580, Brazil; jecel@merlintec.com (J.M.d.A.J.); hugo.puertas@ufabc.edu.br (H.P.d.A.)

² Academic Unit of Cabo de Santo Agostinho (UACSA), Federal Rural University of Pernambuco (UFRPE), Cabo de Santo Agostinho 54518-430, Brazil; oswaldo.ando@ufrpe.br

* Correspondence: mario.gazziro@ufabc.edu.br

Abstract: This work introduces a novel custom-designed 16-bit RISC-V processor, intended for educational purposes and for use in low-resource equipment. The implementation, despite providing registers of 16 bits, is based on RV32E RISC-V ISA, but with some key differences like a reduced instruction set that is optimized for embedded systems, the removal of floating-point instructions, reduced register count, and modified data types. These changes enable the processor to operate efficiently in resource-constrained environments while still maintaining assembly-level compatibility with the standard RISC-V architecture. The educational aspects of this project are also a key focus. By working on this project, students can gain hands-on experience with digital logic design, Verilog programming, and computer architecture. The project also includes tools and scripts to help students transform assembly code into binary format, making it easier for them to test and verify their designs. Additionally, the project’s open-source nature allows for collaboration and the sharing of knowledge among students and researchers worldwide.

Keywords: soft core; processor; RISC-V

1. Introduction

Many authors have made contributions to the study of educational platforms for teaching RISC-V architectures [1–10]. Sallar et al. [11] believes that the availability of a modern, accessible, and FPGA-friendly RISC-V RTL implementation together with a corresponding Virtual Prototypes (by using emulators) configuration would be very beneficial for the academic community to stimulate further research and for educational purposes, as proposed in their MicroRV32 framework presented in Figure 1.



Citation: de Assumpção, J.M., Jr.; Ando, O.H., Jr.; de Araújo, H.P.; Gazziro, M. An Educational RISC-V-Based 16-Bit Processor. *Chips* 2024, 3, 395–407. <https://doi.org/10.3390/chips3040020>

Received: 9 October 2024

Revised: 18 November 2024

Accepted: 28 November 2024

Published: 30 November 2024



Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

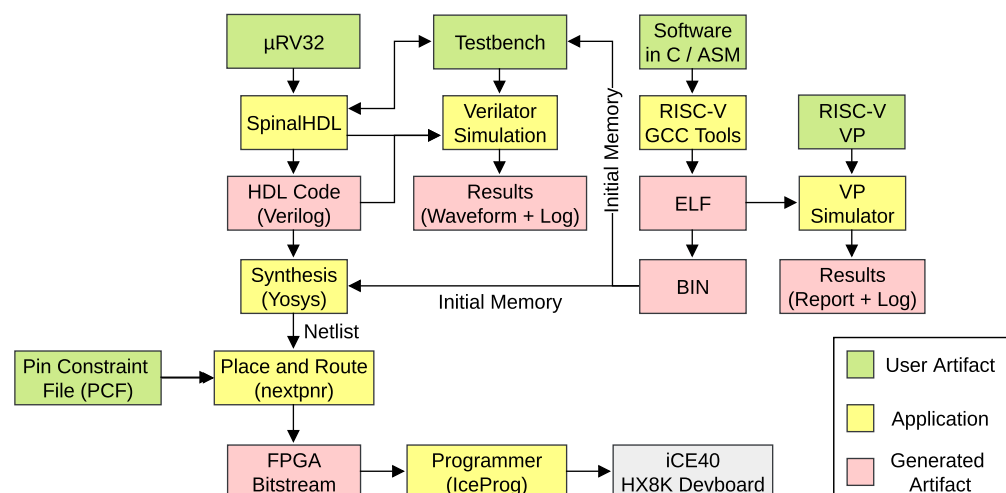


Figure 1. Overview of the open-source HW/SW simulation and design tool flow of MicroRV32 [11].

In order to simplify this approach, we develop the **drv16** processor HW (using the existing SW framework) as a custom implementation based on the RISC-V standard [12], with a distinct characteristic being its reduced register count of 16×16 -bit registers. While it implements a subset of instructions from RV32E, those that are supported share the same mnemonic and functionality. This design choice enables **drv16** to be effectively utilized as a helper processor in various applications, such as abstracting interfaces to peripherals like keyboards or SD cards.

By reducing the number of registers and instructions, the **drv16** processor achieves significant area savings compared to its RV32E counterpart. In particular, its 16-bit architecture should result in a design that occupies half the area required by an RV32E processor, with even greater reductions possible relative to an RV32I processor [13]. Given the modest memory requirements of many applications using **drv16**, the ability to address more than 64 KB would indeed be unnecessary.

An additional motivation for reducing the state is to make it easier for people to handle it. A 16-bit number like 0xC7F0 is more digestible than something like 0xC7F0AA35, which is important in an educational context. An advanced digital design and simulation tool named DIGITAL, developed by Neemann [14], is employed for the design process.

2. Instruction Set

The design prioritizes a compact implementation while maintaining an acceptable performance for a multi-cycle processor. The instruction set is executed in a two-clock cycle sequence, comprising a *fetch* and *execute* phase. In cases where an immediate extension word is present, an additional clock cycle is required, resulting in a maximum of three clock cycles per instruction. Furthermore, the presence of multiple extensions can lead to longer execution times. Notably, memory regions consisting entirely of zeros execute a series of extensions at a rate of one clock cycle per word.

The clock frequency is limited by the critical path, which makes this processor slower than a pipelined one, as seen in Equation (1):

$$\begin{aligned} \text{ClockCycle} > IR\text{delay} + \text{ControlUnitDelay} + \text{RegisterBankDelay} + \\ & \text{ALUinputMuxDelay} + \text{ALUdelay} + \text{MemoryDelay} + \\ & \text{byteMuxDelay} + \text{RegisterInputMuxDelay} \end{aligned} \quad (1)$$

The **drv16** processor employs a binary encoding scheme for its instructions, consisting of 16-bit words, incompatible with the RISC-V C extension. The most significant difference is the instruction which appends 12 bits to the 4-bit immediate value of the following instruction, effectively treating it as a unified 32-bit operation. All other instructions have the following format:

15 14 13 12	11 10 09 08	07 06 05 04	03 02 01 00
rD	rS1	rS2	operation

Register x0 holds the current program counter (PC), but when the rD field is zero, no register is changed, and when rS1 or rS2 is zero, the value 0 is used in place of whatever is in x0. Using a non-latched memory, which is an option for BRAMs in some FPGAs and is how external SRAM chips work, all instructions can run in two clock cycles (fetch and execute). The PC holds the address of the currently executing instruction during the execute phase, so the address presented during fetch is the result of adding two to the PC. This has a side effect that the offset for **JAL** and the branches is from the current instruction and not the next one like in RISC-V. This is compensated for in the assembler to avoid complicating the hardware. Since we have 16-bit instead of 12-bit offsets of the RISC-V, this is not a limitation. The offset for **JALR** does not need any changes since the PC does not enter into its calculation.

In Table 1, @rS2 indicates the 16-bit value in the register addressed by the rS2 field of the instruction while a plain rS2 indicates a 4-bit immediate value extended to 16 bits using the start of a 32-bit instruction pair. rD indicates a four-bit immediate value that may or may not be extended, depending on whether the previous instruction forms part of a pair.

Table 1. drv16 operation codes, mnemonics, execute, and fetch control.

Operation	Mnemonic	Execute	Fetch
0			@IM := @IR, @IR := mem[@PC := @PC + 2]
1	JAL	@rD := @PC + 2	@IR := mem[@PC := @PC + (@IM rS2)]
1	JALR	@rD := @PC + 2	@IR := mem[@PC := @rS1 + (@IM rS2)]
2	BEQ	cond := @rS1 = @rS2	@RI := mem[@PC := @PC + (cond?(@IM rD):2)]
2	BNE	cond := @rS1 ~ = @rS2	@RI := mem[@PC := @PC + (cond?(@IM rD):2)]
3	BLT	cond := @rS1 < @rS2	@RI := mem[@PC := @PC + (cond?(@IM rD):2)]
3	BGE	cond := @rS1 >= @rS2	@RI := mem[@PC := @PC + (cond?(@IM rD):2)]
4	LB	@rD := SignExtend(mem[@rS1 + (@IM rS2)])	@IR := mem[@PC := @PC + 2]
5	LH	@rD := mem[@rS1 + (@IM rS2)]	@IR := mem[@PC := @PC + 2]
6	SB	mem[@rS1 + rD] := 8Bits(@rS2)	@IR := mem[@PC := @PC + 2]
7	SH	mem[@rS1 + rD] := @rS2	@IR := mem[@PC := @PC + 2]
8	LBU	@rD := ZeroExtend(mem[@rS1 + (@IM rS2)])	@IR := mem[@PC := @PC + 2]
9	ADD	@rD := @rS1 + @rS2	@IR := mem[@PC := @PC + 2]
9	ADDI	@rD := @rS1 + (@IM rS2)	@IR := mem[@PC := @PC + 2]
A	SUB	@rD := @rS1 - @rS2	@IR := mem[@PC := @PC + 2]
A	SUBI	@rD := @rS1 - (@IM rS2)	@IR := mem[@PC := @PC + 2]
B	SLT	@rD := @rS1 < @rS2	@IR := mem[@PC := @PC + 2]
B	SLTI	@rD := @rS1 < (@IM rS2)	@IR := mem[@PC := @PC + 2]
C	SRS	@rD := (@rS1 >> 1) (@rS2 & 0x8000)	@IR := mem[@PC := @PC + 2]
C	SRSI	@rD := (@rS1 >> 1) (@IM & 0x8000)	@IR := mem[@PC := @PC + 2]
D	AND	@rD := @rS1 & @rS2	@IR := mem[@PC := @PC + 2]

Table 1. Cont.

Operation	Mnemonic	Execute	Fetch
D	ANDI	@rD := @rS1 & (@IM rS2)	@IR := mem[@PC := @PC + 2]
E	OR	@rD := @rS1 @rS2	@IR := mem[@PC := @PC + 2]
E	ORI	@rD := @rS1 (@IM rS2)	@IR := mem[@PC := @PC + 2]
F	XOR	@rD := @rS1 ^@rS2	@IR := mem[@PC := @PC + 2]
F	XORI	@rD := @rS1 ^(@IM rS2)	@IR := mem[@PC := @PC + 2]

Most instructions have two variations, and the presence or not of the extension selects between them. In the case of **BEQ** and **BNE**, it is the least significant bit of **rD** (extended or not) that selects between them as the bit would otherwise be wasted since we cannot branch to odd addresses. The same trick is used to select between **JAL** and **JALR**.

The **drv16** processor includes the **SUBI** instruction, which is not present in RV32E due to the latter's ability to handle negative constants for ADDI operations. Notably absent from **drv16** are unsigned comparison instructions (**SLTIU**, **SLTU**, **BLTU**, and **BGEU**). Additionally, **LUI** and **AUI** instructions are missing, as constants larger than 12 bits require different generation mechanisms in the **drv16** architecture. The hardware to implement shifts can be very large when compared to the rest of the processor, so the shift operations (**SLLI**, **SRLI**, **SRAI**, **SLL**, **SRL**, and **SRA**) were also omitted. But, **SLLI x3,x4,3** can be implemented using the sequence **ADD x3,x4,x4 . ADD x3,x3,x3 . ADD x3,x3,x3**. Right shifts are implemented using the **SRS** (shift right step) instruction that is not a RV32E one.

So, **SRAI x3,x4,3** can be implemented as the sequence **SRS x3,x4,x4 . SRS x3,x3,x3 . SRS x3,x3,x3** while **SRLI x3,x4,3** can become **SRS x3,x4,zero . SRS x3,x3,zero . SRS x3,x3,zero**. The **SRSI** instruction is present to simplify the hardware, but is not as useful. **ECALL** and **EBREAK** are the two remaining RV32E instructions missing from **drv16**.

3. Implementation

The project *system.dig* (shown in Figure 2) includes the **drv16** processor connected to an asynchronous RAM with 32K words of 16 bits each. Address 0xFFFF (word address 0x7FFF) is also mapped to the terminal for writes and a keyboard interface for reads. The *reset* signal must be held high for at least four clock cycles to force execution to start at the instruction at address 0x0000 in memory.

Two complications that RISC-V shares with **drv16** relative to some simpler processors are the byte access to memory and the special treatment of register zero. This is further complicated in **drv16** by storing the program counter in the register bank's address zero since that would have been otherwise unused. This also allows the adder in the ALU to also increment the program counter.

The bottom third of Figure 3 shows how byte reads are handled on the left and how byte writes are dealt with on the right. For word reads, the data just pass straight through, while for byte reads, either the top or bottom byte of the incoming data (depending on the address bit 0) becomes the bottom byte to be written to the register, and the top byte is either all zeros or copies of the sign bit from the bottom byte (depending on whether **LBU** or **LB** is being executed).

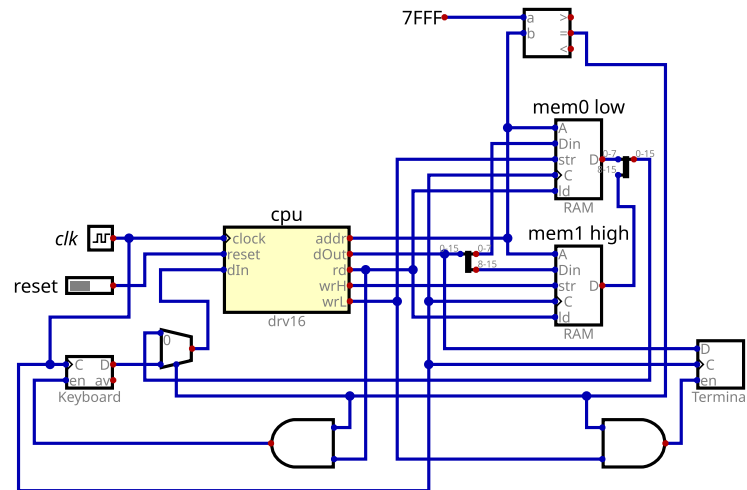


Figure 2. Complete CPU system using **drv16** soft core processor. The blue lines are the data buses, with small blue symbols indicating inputs, small red symbols indicating outputs and the big blue symbols indicating bus intersections.

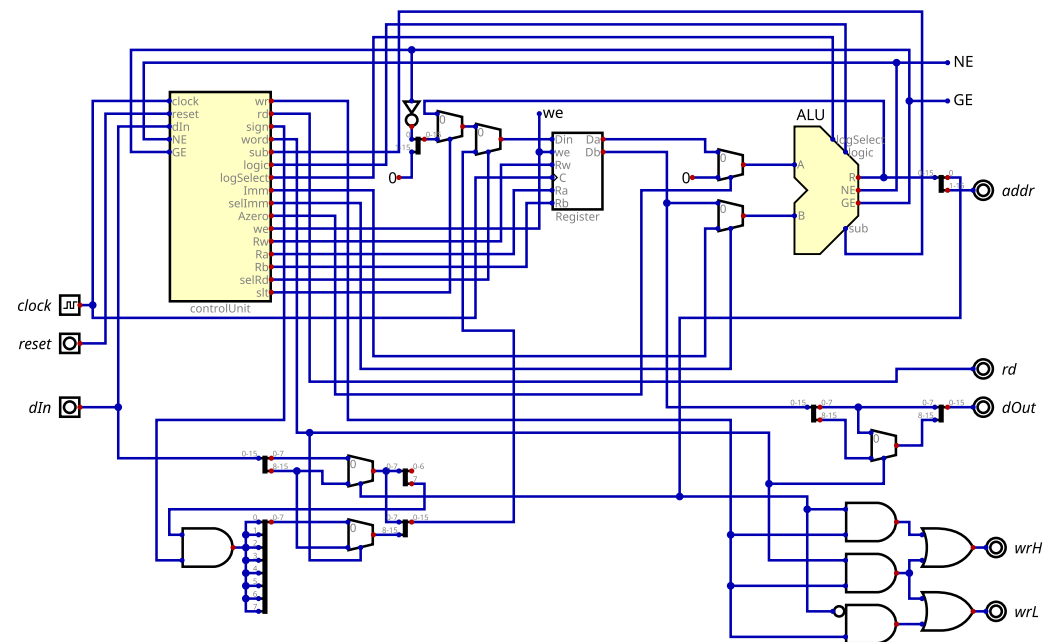


Figure 3. Implementation of the **drv16** soft core processor.

For byte writes, the bottom eight bits from the register are output twice and the corresponding signal (*wrL* or *wrH*) makes it so that one is used and the other ignored. For word writes, the data just pass through unchanged.

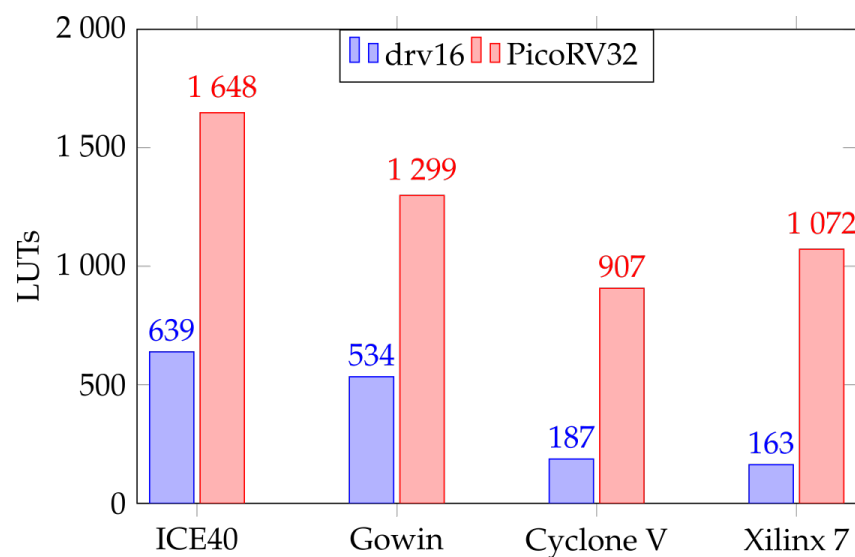
4. FPGAs and ASICs

Exporting the subcomponents of **drv16** as Verilog, as well as a circuit with only the register file, and using the Yosys [15] tool to implement them for various FPGAs and integration technologies, it is possible to make some comparisons as one has in Table 2. In that table, the presented numbers stand for the quantity of registers/LUTs/math/DMBs (distributed memory blocks) for each technology implementation.

Table 2. Subcomponents of **drv16** as Verilog.

Technology	Registers	ALU	execPLA	Control	Total: drv16
NANDs	5029	660	32	628	6534
ICE40	256/373/0/0	0/99/16/0	0/12/0/0	33/58/0/0	289/639/16/0
Gowin	0/0/0/8	0/145/17/0	0/12/0/0	33/69/0/0	33/534/17/8
Cyclone V	0/0/0/32	0/55/18/0	0/11/0/0	33/51/0/0	33/187/18/32
Xilinx 7	0/0/0/8	0/50/5/0	0/11/0/0	33/49/0/0	33/163/5/8

Figure 4 presents a comparison between **drv16** and RV32E-based **PicoRV32** resource usage defined by [13].

**Figure 4.** Comparison of LUTs usage between the **drv16** and its analog **PicoRV32**.

5. Datapath

The main blocks of the datapath are the register bank and the ALU. The multiplexers of the byte memory access adaptor were previously described. Two more multiplexers allow the data written back to the destination register to be either the ALU result, dIn from memory, or a boolean value indicating a signed Less Than result for a comparison. Another multiplexer allows the A input of the ALU to either come from the registers or be the constant 0. In the same way, another multiplexer allows the B input of the ALU to either come from the registers or the immediate value (which can be forced to the constant 0 or the constant 2 inside the control unit).

ALU

The ALU is presented in Figure 5. Looking at all instructions, we need to be able to add and subtract a pair of 16-bit numbers; perform a bitwise *AND*, *OR*, and *XOR* operations between them; and also handle the odd shift to the right combining with a bit from the other operand. When subtracting, we need to indicate the signed comparisons $A \geq B$ and $A \neq B$.

Two odd-looking parts of the ALU are what seem to be many shorted wires at the very top and a large number of OR gates taking up the whole right half. The first is just an expansion of the input signal *sub* from 1 bit to 16 bits (all with the same value, which is why they are shorted) so it can be combined with the 16-bit-wide input *B* in an *XOR* gate to invert it for subtractions. The 16-bit *OR* gate is shown as a tree of five four-input *OR* gates,

but the logic synthesis tools are free to implement that in a way that is optimized for the different technologies. It simply detects when the ALU output is 16 zero bits.

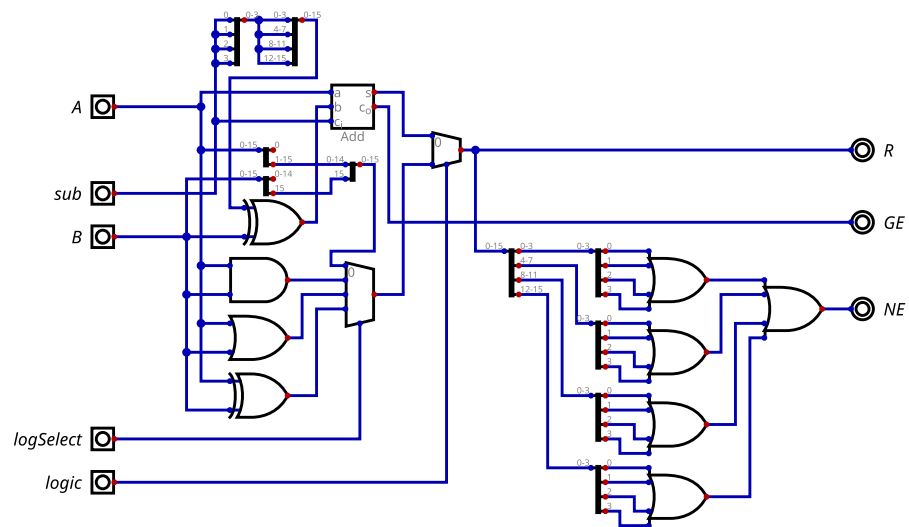


Figure 5. ALU (Arithmetic Logic Unit).

6. Control Unit

The control unit uses the following inputs to perform its job: *clock*, *reset*, *dIn*, *NE*, and *GE*. It generates the following: *wr*, *rd*, *sign*, *word*, *sub*, *logic*, *logSelect*, *Imm*, *selImm*, *Azero*, *we*, *Rw*, *Ra*, *Rb*, *selRd*, and *slt*. There are also some internal signals such as *alt*, *even*, *const2*, *selConst*, and *immLow*.

The very simple circuit in Figure 6 helps the control unit handle register zero. It allows any of the three instruction fields to be overridden by the PC and indicates if special handling (replace with 0 for the sources and do not write for the destination) is needed.

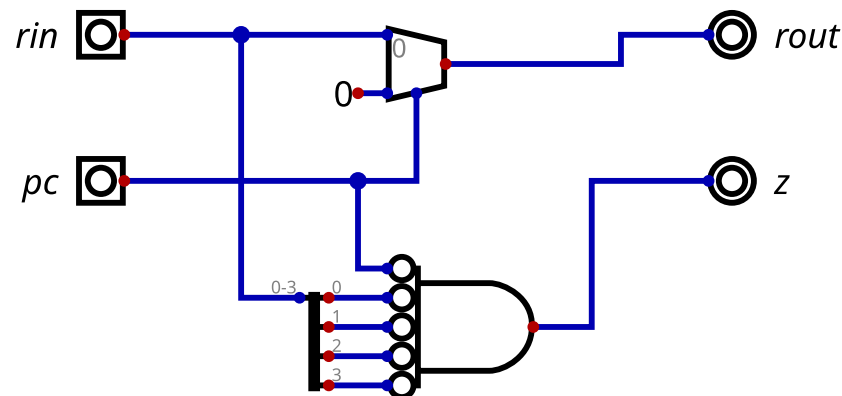


Figure 6. Register x0 handling.

When field *rD* is 0, then the result should not be saved to the register ($we := 0$), unless it is (perhaps also) being forced to the PC, which happens during any fetch. When field *rS1* is 0, then input *A* of the ALU must be forced to zero ($Azero := 1$) unless it is being forced to be the PC. Combined with what was said about reset, we have $Azero := reset \mid (zS1 \&!aPC)$. When field *rS2* is 0 and is not immediate, then input *B* of the ALU must be forced to zero. This field is never forced to be the PC, which is why a four-input *AND* gate is used in place of the full helper circuit. Activating the *selImm* and *selConst* signals but not *even* (a combination also used by *reset*) will do the job.

Table 3 shows **drv16** control unit signals for both the execute and fetch phases of different instructions. The correspondence with Table 1 is slightly obscured since the latter

lists instructions vertically and the former horizontally, but they are two ways of describing the same behavior.

Register *IR* saves the instruction read from memory during the fetch cycle and *IM* saves the previous value of the top 12 bits for *IM* or is cleared to 0 if the previous cycle was an execute. This allows instructions that do not depend on the prefix to indicate an immediate to not require a prefix when that would just be 0x0000 (but the current assembler does not take advantage of that).

The single-bit *fetch* flip-flop is the heart beat of the processor. Its normal output indicates a fetch cycle while its inverted output indicates an execute cycle. A fetch can follow an execute or another fetch where the data coming from memory will be a prefix instruction. If *reset* is active, then the processor will be stuck fetching from the memory location 0x0000. During *reset*, the signals to execute `@IR := mem[@PC := 0]` are active. With both inputs forced to zero during *reset*, the values of *logic* and *logSelect* do not make a difference.

When the instruction was **JAL**, **JALR**, or a branch with `cond == true`, then some signals are changed in the next fetch cycle.

Branches are the only case where no register is written to as the result is saved in *cond* instead. *IR0* selects between *GE* and *NE* inputs and *alt* inverts the test. *alt* is just the lowest bit of the immediate value before it is optionally cleared by *even*.

Table 3. *drv16* signals.

	Execute		Fetch
<i>inputs:</i>		<i>inputs:</i>	
IR3	000000111111111	reset	100000
IR2	001111000000011	cond	XXXX01
IR1	0100110001111XX	IR1	XX0011
IR0	1X01010110011XX	alt	XX01XX
imm	XXXXXXXX01010101	JorB	X01111
<i>outputs:</i>		<i>outputs:</i>	
even	1X00000X0X0X0X0	even	X11111
const2	1XXXXXXXXXXXXXX	const2	01XX1X
selImm	101111101010101	selImm	1
selConst	1X00000X0X0X0X0	selConst	110010
Azero	0	Azero	100000
sub	010000001111XX	sub	0
logic	000000000000011	logic	X00000
aPC (force A to PC)	100000000000000	aPC	X11011
lowImm	0X00110X0X0X0X0	lowImm	0X0011
rd	001100100000000	rd	1
wr	000011000000000	wr	0
we	101100111111111	we	1
sign	XX1XXX0XXXXXXXX	sign	X
word	XX01010XXXXXXXX	word	X
selRd	0X11XX100000000	selRd	0
slt	0XXXXXX00001100	slt	0

We can use the DIGITAL function to generate a circuit from this table. An “X” in an input means that there are actually two columns—one with this value as “0” and another with this value as “1”. With five inputs, the truth table will have 32 entries. An “X” in an output means that either a “0” or a “1” are acceptable and DIGITAL can generate a smaller circuit by selecting one or the other.

The block is named “PLA” (Programmable Logic Array) in this project because that would be a normal way of implementing such circuits in early integrated circuits. A PLA implements a “sum of products” combinational logic (ORs that have as inputs ANDs connected to some of their inputs or their inverses) in a very compact layout. If a tool does not have a PLA layout generator, then standard cells will produce the same result but with a larger area. In FPGAs, these circuits can be implemented with just one or a few LUTs (LookUp Tables) for each output.

The exec PLA outputs in control unit are multiplexed with the explicit circuit for the *fetch* half, seen at the very bottom of Figure 7. Since only three signals are not constant in the *fetch* half of the table, only the *exec* half is generated as the block of Figure 8.

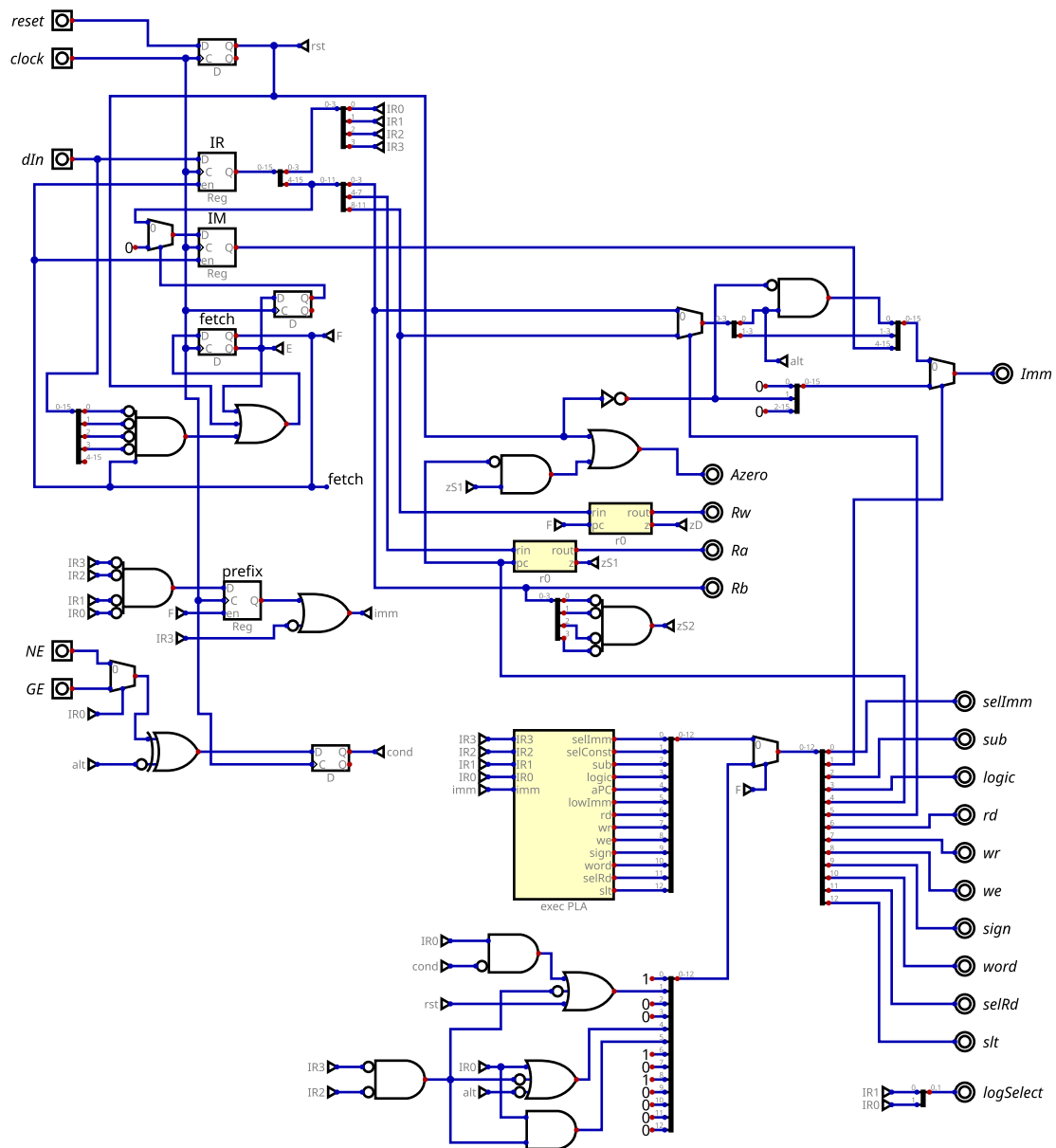


Figure 7. Control unit of *drv16* soft core processor.

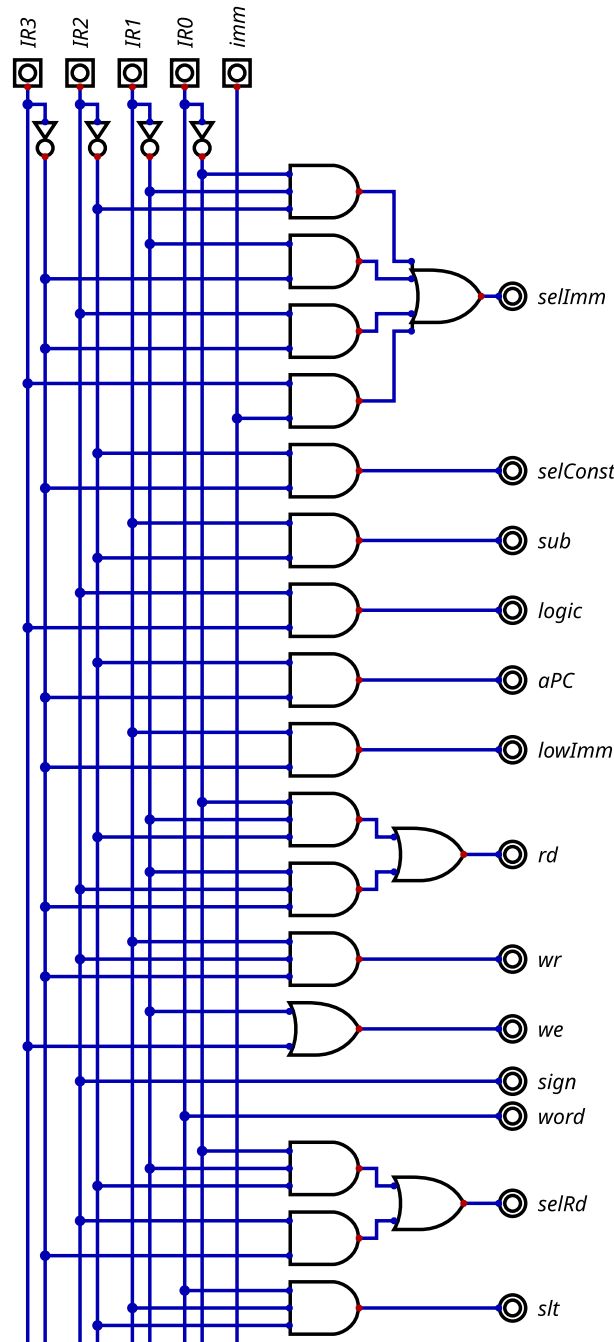


Figure 8. Execution PLA (Programmable Logic Array).

7. Software

It is possible to use the GNU AS assembler, even if it is for a processor like the x86, to generate binaries for **drv16**. Macros and other definitions in *drv16.inc* allow any assembly program that includes it to use all the instructions defined above. One limitation is that symbols cannot be used directly, so if a label `width` is defined somewhere, its data must be read with `lh x4, zero, (width-absStart)`. File *drv16.inc* defines `absStart` as address 0. The macro for pseudoinstruction LA does this internally.

A second limitation is that while the hardware does not require a prefix for memory and control flow instructions that have immediate values of 15 or less, *drv16.inc* generates a useless `0x0000` prefix anyway. A dedicated assembler doing more than one pass would typically make programs around 25% smaller (which also eliminates clock cycles).

A bash script, *as2hex*, will transform an assembly source file *.S* into an Intel Hex equivalent to the binary. DIGITAL can load such files directly into a memory block before the start of a simulation.

The program *gcd.S* calculates the greatest common denominator between two numbers that are built into the sources (currently 12 and 18). A message with the result and the two numbers is printed in the terminal window. The code to print strings and decimal numbers (always three digits with leading zeros) dwarfs the actual GCD part.

Figures 9 and 10 show the internal operations of the *drv16* soft core processor evaluating a greatest common denominator of 012 and 018 and giving 006 as the result in the plain text output on the DIGITAL Terminal.

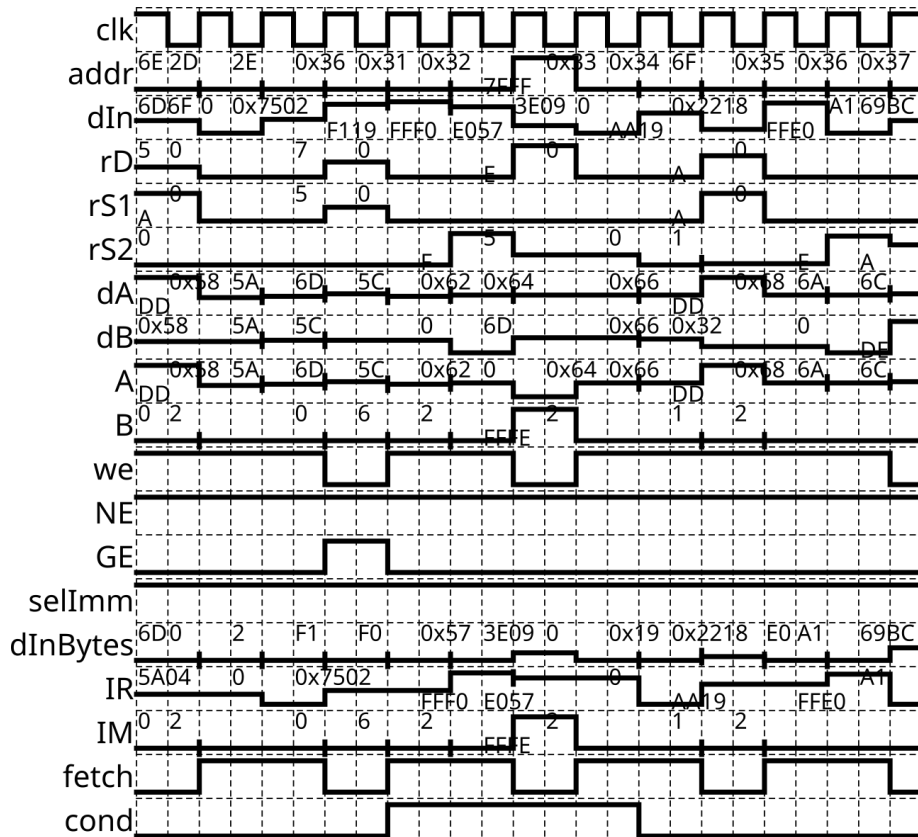


Figure 9. Internal operations timing diagram of *drv16*.

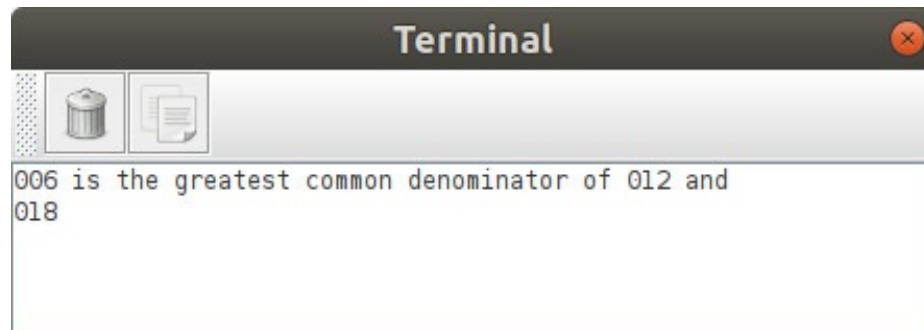


Figure 10. Terminal output in plain text.

Another program was developed to test the fixed-point arithmetic capabilities of the *drv16* processor. A *mandelbrot.S* presented in Figure 11 demonstrates a text version of the famous fractal.

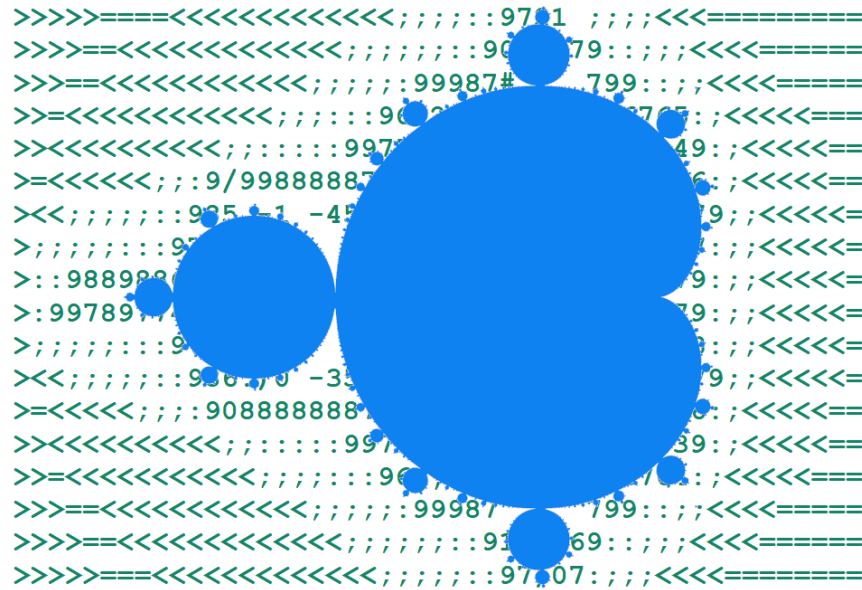


Figure 11. Mandelbrot-type fractal generated as text output using **drv16** emulation on *DIGITAL* tool and its overlay with the high resolution graphical model.

8. Performance

A benchmark Table 4 was created to evaluate the performance of the **drv16** processor against other CPUs (RISC-V and non-RISC-V) using Gowin FPGA executing a simple sine wave generator using only *shifts*.

Table 4. CPU benchmark.

	drv16	MCPU16 [16]	DarkRISCV [17]
Gowin (LUTs)	282	69	1431
Gowin (FFs)	33	48	176
Gowin (FMax)	95 MHz	313 MHz	76 MHz
Gowin (total power)	140 mW	138 mW	178 mW
Gowin (dynamic power)	19 mW	17 mW	57 mW
sine (lines of code)	62	129	57
sine (bytes of code)	164	2403	128
sine (clocks to 50 sin points)	23,118	130,831	9360
sine (per second)	4109 Hz	2392 Hz	8120 Hz

9. Conclusions

The goal of using less than half of the FPGA resources compared to RV32E implementations like PicoRV32 was achieved. The implementation in the form of an RTL-style schematic provides another option for educators who want to teach computer architecture using classical digital logic techniques instead of HDL languages only. The final performance, when compared to a pipelined CPU like the DarkRISCV [17], was very good as it achieved half the performance-generating sine waves at half the (dynamic) energy cost using a very similar clock.

Author Contributions: The conceptualization and design was done by J.M.d.A.J. The data curation was provided by O.H.A.J. The validation was performed by H.P.d.A. and the project coordination was realized by M.G. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Informed Consent Statement: Not applicable.

Data Availability Statement: All data are available at [18].

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Giorgi, R.; Mariotti, G. WebRISC-V: A Web-Based Education-Oriented RISC-V Pipeline Simulation Environment. In Proceedings of the 46th Annual International Symposium on Computer Architecture, WCAE@ISCA, Phoenix, AZ, USA, 22 June 2019. [CrossRef]
2. Harris, S.L.; Chaver, D.; Piñuel, L.; Pérez, J.I.G.; Liaqat, M.; Kakakhel, Z.L.; Kindgren, O.; Owen, R. RVfpga: Using a RISC-V Core Targeted to an FPGA in Computer Architecture Education. In Proceedings of the International Conference on Field-Programmable Logic and Applications, Dresden, Germany, 30 August–3 September 2021. [CrossRef]
3. Morgan, F.; Beretta, A.; Gallivan, I.; Clancy, J.; Rousseau, F.; George, R.; Bako, L.; Callaly, F. RISC-V Online Tutor. In *Online Engineering and Society 4.0. REV 2021; Lecture Notes in Networks and Systems*; Springer: Cham, Switzerland, 2021. [CrossRef]
4. Nitta, C.; Kaloti, A.; Wang, S. RISC-V Console: A Containerized RISC-V Based Game Console Emulator for Education. In Proceedings of the Annual Conference on Innovation and Technology in Computer Science Education, Dublin, Ireland, 8–13 July 2022. [CrossRef]
5. McGrew, T.; Schonauer, E. Framework and Tools for Undergraduates Designing RISC-V Processors on an FPGA in Computer Architecture Education. In Proceedings of the 2019 International Conference on Computational Science and Computational Intelligence (CSCI), Las Vegas, NV, USA, 5–7 December 2019. [CrossRef]
6. Meeradevi, T.; Kumar, M.; Mourish, B.M.; Sivaa, V.S.; Samikannu, R.; Sasikala, S. Design of 32 Bit RISC V Processor. In Proceedings of the International Conference on Computing Communication and Networking Technologies, Kamand, India, 24–28 June 2024. [CrossRef]
7. Poli, L.; Saha, S.; Zhai, X.; McDonald-Maier, K. Design and Implementation of a RISC V Processor on FPGA. In Proceedings of the International Conference on Mobile Ad-hoc and Sensor Networks, Exeter, UK, 13–15 December 2021. [CrossRef]
8. Jamieson, P.; Le, H.; Martin, N.; McGrew, T.; Qian, Y.; Schonauer, E.; Ehret, A.; Kinsy, M.A. Computer Engineering Education Experiences with RISC-V Architectures—From Computer Architecture to Microcontrollers. *J. Low Power Electron. Appl.* **2022**, *12*, 45. [CrossRef]
9. Oruç, A.; Atmaca, A.; Sengun, Y.N.; Yenyol, A.S. CodeAPEel: An Integrated and Layered Learning Technology For Computer Architecture Courses. *arXiv* **2021**, arXiv:2104.09502.
10. Minev, P.; Kukenska, V.; Varbov, I.; Dinev, M. A Practical Computer Architecture Education with RISC-V and TL-Verilog. In Proceedings of the 2023 XXXII International Scientific Conference Electronics (ET), Sozopol, Bulgaria, 13–15 September 2023. [CrossRef]
11. Ahmadi-Pour, S.; Herdt, V.; Drechsler, R.; Ahmadi-Pour, S.; Herdt, V.; Drechsler, R. The MicroRV32 framework: An accessible and configurable open source RISC-V cross-level platform for education and research. *J. Syst. Archit.* **2022**, *133*, 102757. [CrossRef]
12. SiFive. The RISC-V Instruction Set Manual. Volume I: User-Level ISA, Version 2.2. 2017. Available online: <https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf> (accessed on 4 September 2024).
13. Gazziro, M.; Junior, J.; Junior, O.; Cavallari, M.; Carmo, J. Design and Evaluation of Open-Source Soft-Core Processors. *Electronics* **2024**, *13*, 781. [CrossRef]
14. Neemann, H. DIGITAL—A Digital Logic Designer and Circuit Simulator. Github. Available online: <https://github.com/hneemann/Digital> (accessed on 4 September 2024).
15. Shah, D.; Hung, E.; Wolf, C.; Bazanski, S.; Gisselquist, D.; Milanovic, M. Yosys+nextpnr: An Open Source Framework from Verilog to Bitstream for Commercial FPGAs. In Proceedings of the 2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), Los Alamitos, CA, USA, 28 April–1 May 2019; pp. 1–4. [CrossRef]
16. Tim. MCPU—Minimal CPU for a 32 Macrocell CPLD. Github. Available online: <https://github.com/cpldcpu/MCPU> (accessed on 16 November 2024).
17. Samsoniuk, M. DarkRISCV Processor. Github. Available online: <https://github.com/darklife/darkriscv> (accessed on 16 November 2024).
18. Assumpcao, J., Jr. drv16. Github. Available online: <https://github.com/jeceljr/digitalCPUzoo/tree/main/drv16> (accessed on 17 November 2024).

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.