Novel Toolset for Efficient Hardwired Micro-Op Translation in Embedded Microarchitectures

Kevin J. Phillipson, Michael G. Rywalt, Baibhab Chatterjee, Eric M. Schwartz, Greg Stitt
Department of Electrical & Computer Engineering, University of Florida, Gainesville, Florida, USA
team@turbo9.org – www.turbo9.org

***Abstract*—Modern SoCs require increasingly complex embedded control deep within their numerous sub-blocks without adding significant die area. This motivated the creation of μRTL, a novel toolset for systematically designing efficient pipelined implementations of embedded instruction sets originally intended for multi-cycle execution. μRTL utilizes hardwired micro-op translation, a technique commonly used in the instruction decoders of large super-scalar microprocessors, however this technique has been overlooked for designing smaller, more efficient embedded microprocessors. Furthermore, the tools to develop instruction decoders with micro-op translation are proprietary and the techniques are trade secrets. The μRTL toolset is open-source and this paper clearly presents the methodology. The methodology emphasizes direct opcode decoding from multiple synthesized Verilog blocks vs traditional microprogramming which uses sequential decoding from a ROM. Our results show that a pipelined μRTL microarchitecture achieves a 21.8% reduction in size compared to a hardwired multi-cycle implementation of the same instruction set. Additionally, the performance of 0.75 DMIPS/MHz surpasses the RISC-V PicoRV32 by 44.2% and the AVR RISC by 82.9%. These improvements in performance, power, and area are of interest to embedded system architects.**

***Index Terms*—micro-op, pipelining, instruction decode, micro-assembler, microcode, embedded microarchitecture**

I. Introduction

Efficient micro-op translation is a well-established technique used in the decode stage of large high-end pipelined processors. In this paper, we demonstrate the significant advantages of applying this technique to an embedded instruction set architecture (ISA) originally intended for multi-cycle execution, enabling a pipelined implementation that is powerful yet area efficient. However, designing such translation logic is challenging and the tools and methodology are not publicly available. To assist with these challenges, we introduce the open-source μRTL toolset for designing embedded instruction decoders utilizing micro-op translation.

The pipelined μRTL microarchitecture achieves a 21.8% reduction in size compared to the multi-cycle microarchitecture implementing the same ISA. Additionally, the performance of the μRTL microarchitecture at 0.75 DMIPS/MHz surpasses the RISC-V PicoRV32 by 44.2%, the AVR RISC by 82.9%.

II. Background

Research into area-optimized architectures often creates innovative new ISAs [1]. However, they are often impractical, lack software tools or an established code base. Our approach is to select an existing, well-supported, accumulator-based ISA and create a pipelined implementation utilizing micro-op translation. Intel first created a pipelined micro-op translation architecture to compete with the performance of RISC architectures [2]. Since then, similar techniques have been applied to many large high-end processors but overlooked for embedded architectures [3], [4]. The following provides the necessary background before presenting our μRTL methodology.

A. Instruction Decoder Design

Fundamentally, a microprocessor integrates a data path and a control unit to execute instructions. The former includes internal registers and the ALU to handle data, while the latter decodes instructions into control signals that guide the data path through the necessary micro-states. The control unit is implemented as a single state machine in a multi-cycle design, or as overlapping stages in a pipelined design [5]. This section focuses on three methods for decoding control signals from the instruction opcode and micro-state, as shown in Figure 1.

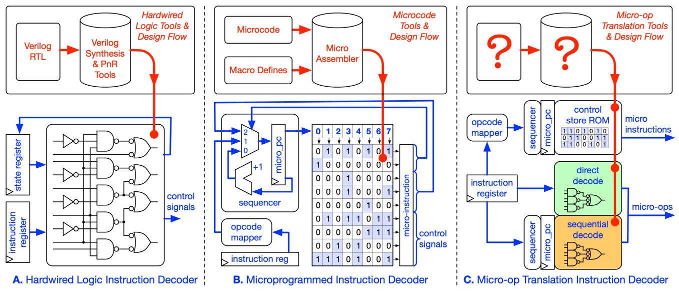


Fig. 1. Instruction Decoders: Hardwired, Microprogram, Micro-op Translation

1) Hardwired Logic Instruction Decoder: Shown in Figure 1A is a hardwired control implementation. In this case, the control signals are derived directly through optimized combinational logic. This design typically results in faster clock speeds and potentially lower latencies for instruction execution. The architecture is fixed, making it efficient for its intended set of instructions, but not easily updated or extended without re-synthesis. Hardwired control is commonly implemented using a hardware description language such as Verilog. However, the large task of decoding an ISA in such a language presents a challenge in achieving a balance between optimal decoding, design clarity, and ease of maintenance.

2) Microprogrammed Instruction Decoder: Microprogrammed control, often associated with the term "control store", decouples the instruction decoding from the hardware logic [6]. In this case, the control signals are mapped into a ROM or RAM driven by an address sequencer. The primary disadvantage is the separation of control signals from the optimized instruction encoding, using the sequential ROM address to derive them instead. An *opcode mapper* (see Figure 1B) is used to map an instruction to the starting address of a sequence of micro-instructions and often introduces an extra cycle of latency. The structured nature of this implementation is the main advantage, and micro-assembler tools force the designer to methodically break down the control sequences needed to implement a large instruction set.

3) Micro-op Translation Decoder: Many modern instruction decoders include both the higher latency microprogrammed decoder and a highly optimized hardwired decoder feeding control signals into a combined stream of micro-instructions (Figure 1C). This new micro-op translation path leverages the low latency and optimal hardwired logic to decode micro-ops efficiently for commonly used instructions. Significant emphasis is placed on direct decode logic, aiming to extract control signals directly from the opcode's bit fields in order to minimize redundant sequential states. This specialized hardwired decoder must conform to the same structured micro-instruction format as the microprogrammed decoder and derive its micro-ops from the same control signal definitions and opcode bit fields. Therefore, it is desirable to use a similar toolset as the microprogrammed methodology with the same description of the target architecture and similar micro-op mnemonics. This specialized micro-assembler should be capable of deriving the micro-op mnemonic's operands from either direct decoded or sequential decoded control signals.

III. Method: μRTL Based Micro-op Translation

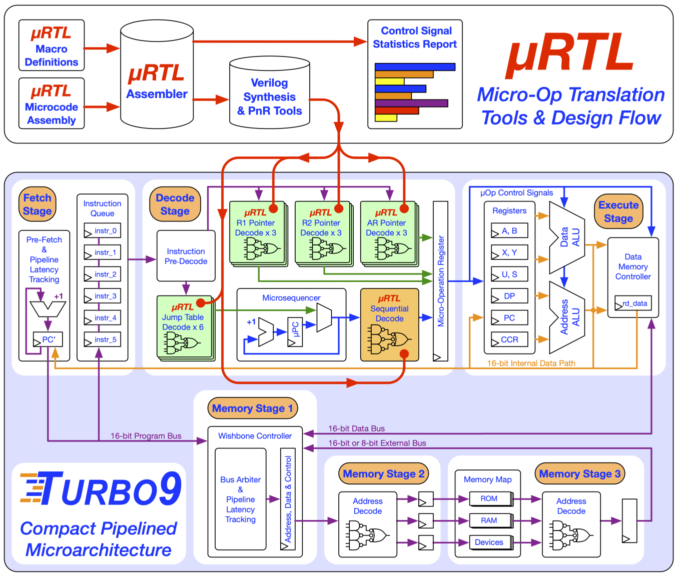


Fig. 2. μRTL Tools and Design Flow with Example Microarchitecture

The μRTL micro-assembler is written in C and can be built and run in a Linux environment. The design flow is shown at the top of Figure 2. In this example, 15 Verilog files are output by μRTL and synthesized into the gates of the hardwired pipelined decode stage. A macro definition input file provides the widths and selectable values of all control signals used to build the micro-op word. It also defines the high-level macros with operand fields that the designer can use to create the mnemonics of the micro-op. The μRTL microcode input file uses these mnemonics with operands pointing to control signals in either the sequential or direct decode logic. Finally, the control signal statistics output is an advanced feature and offers the designer valuable insight to optimize the control signal encoding or the datapath to improve area and timing path delays. This design feedback is critical since we are targeting synthesized logic gates rather than a ROM.

A. Example Embedded μRTL Microarchitecture

The bottom of Figure 2 shows the microarchitecture of the example embedded microprocessor, the Turbo9 [7]. The target applications are area constrained, deeply embedded SoC sub-blocks and mixed-signal ASICs that require high performance [8]. There is also development of an innovative real time operating system, TurbOS, to support these applications [9].

Traditional 32-bit RISC architectures with large register files have excessive area requirements, given that many deeply embedded applications only need 16-bit precision. Therefore, we targeted an 8/16-bit accumulator-based ISA with few internal registers. The Motorola 6809 instruction set was selected for its powerful and efficient features, which align well with the requirements of a C compiler [10]. The second design choice was to implement a pipelined microarchitecture for increased performance. Although the 6809 ISA is much simpler than many RISC ISAs, it violates the RISC load/store principle by using memory as an operand. This creates a challenge in breaking down the instruction addressing mode and data operation to be efficiently pipelined. This motivated the creation of the μRTL micro-op translation methodology, which enabled the development of Turbo9's efficient pipelined microarchitecture. This innovative design approach presents a compelling solution for compact embedded microprocessors.

B. μRTL Micro-op translation example: ADD instructions

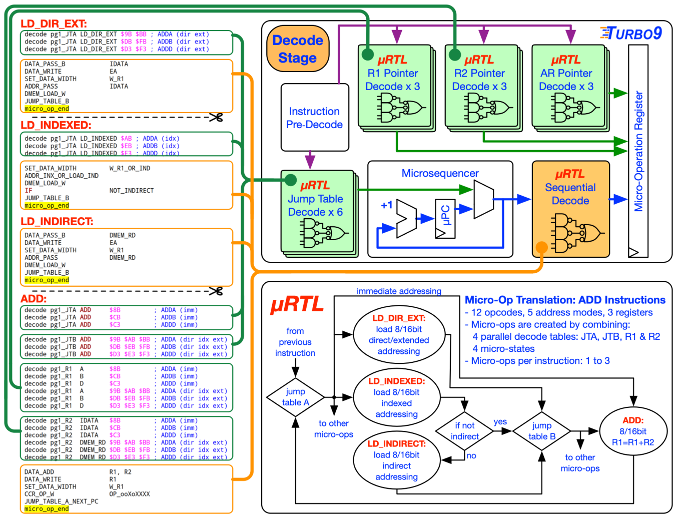
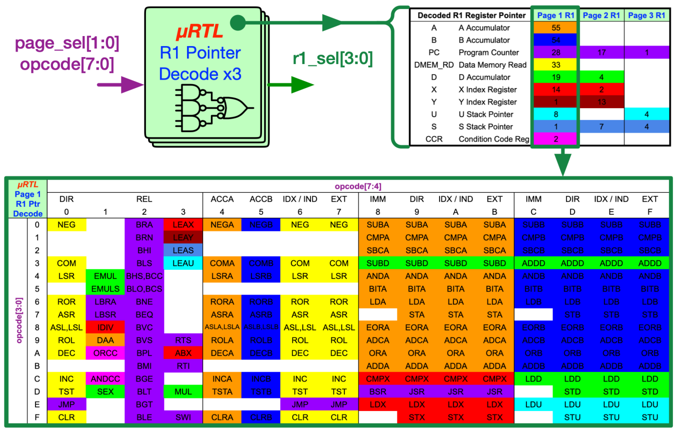

LD_DIR_EXT:
```
decode pg1_JTA LD_DIR_EXT $9B $BB ; ADDA (dir ext)
decode pg1_JTA LD_DIR_EXT $DB $FB ; ADDB (dir ext)
decode pg1_JTA LD_DIR_EXT $D3 $F3 ; ADDD (dir ext)
```
```
DATA_PASS_B          IDATA
DATA_WRITE           EA
SET_DATA_WIDTH       W_R1
ADDR_PASS            IDATA
DMEM_LOAD_W
JUMP_TABLE_B
micro_op_end
```
LD_INDEXED:
```
decode pg1_JTA LD_INDEXED $AB ; ADDA (idx)
decode pg1_JTA LD_INDEXED $EB ; ADDB (idx)
decode pg1_JTA LD_INDEXED $E3 ; ADDD (idx)
```
```
SET_DATA_WIDTH       W_R1_OR_IND
ADDR_INX_OR_LOAD_IND
DMEM_LOAD_W
IF                   NOT_INDIRECT
JUMP_TABLE_B
micro_op_end
```
LD_INDIRECT:
```
DATA_PASS_B          DMEM_RD
DATA_WRITE           EA
SET_DATA_WIDTH       W_R1
ADDR_PASS            DMEM_RD
DMEM_LOAD_W
JUMP_TABLE_B
micro_op_end
```
ADD:
```
decode pg1_JTA ADD     $8B         ; ADDA (imm)
decode pg1_JTA ADD     $CB         ; ADDB (imm)
decode pg1_JTA ADD     $C3         ; ADDD (imm)
```
```
decode pg1_JTB ADD     $9B $AB $BB ; ADDA (dir idx ext)
decode pg1_JTB ADD     $DB $EB $FB ; ADDB (dir idx ext)
decode pg1_JTB ADD     $D3 $E3 $F3 ; ADDD (dir idx ext)
```
```
decode pg1_R1  A       $8B         ; ADDA (imm)
decode pg1_R1  B       $CB         ; ADDB (imm)
decode pg1_R1  D       $C3         ; ADDD (imm)
decode pg1_R1  A       $9B $AB $BB ; ADDA (dir idx ext)
decode pg1_R1  B       $DB $EB $FB ; ADDB (dir idx ext)
decode pg1_R1  D       $D3 $E3 $F3 ; ADDD (dir idx ext)
```
```
decode pg1_R2  IDATA   $8B         ; ADDA (imm)
decode pg1_R2  IDATA   $CB         ; ADDB (imm)
decode pg1_R2  IDATA   $C3         ; ADDD (imm)
decode pg1_R2  DMEM_RD $9B $AB $BB ; ADDA (dir idx ext)
decode pg1_R2  DMEM_RD $DB $EB $FB ; ADDB (dir idx ext)
decode pg1_R2  DMEM_RD $D3 $E3 $F3 ; ADDD (dir idx ext)
```
```
DATA_ADD             R1, R2
DATA_WRITE           R1
SET_DATA_WIDTH       W_R1
CCR_OP_W             OP_ooXoXXXX
JUMP_TABLE_A_NEXT_PC
micro_op_end
```
TURBO9
Decode Stage
Instruction Pre-Decode
µRTL R1 Pointer Decode x 3
µRTL R2 Pointer Decode x 3
µRTL AR Pointer Decode x 3
µRTL Jump Table Decode x 6
Microsequencer
+1
µPC
µRTL Sequential Decode
Micro-Operation Register
µRTL
immediate addressing
from previous instruction
jump table A
to other micro-ops
LD_DIR_EXT: load 8/16bit direct/extended addressing
LD_INDEXED: load 8/16bit indexed addressing
LD_INDIRECT: load 8/16bit indirect addressing
if not indirect
yes
no
jump table B
to other micro-ops
ADD: 8/16bit R1=R1+R2
Micro-Op Translation: ADD Instructions
- 12 opcodes, 5 address modes, 3 registers
- Micro-ops are created by combining:
4 parallel decode tables: JTA, JTB, R1 & R2
4 micro-states
- Micro-ops per instruction: 1 to 3

Fig. 3. μRTL Micro-op Translation Example: ADD Instructions

We encourage the reader to closely study Figure 3, which presents the μRTL microcode implementation of all ADD instruction variations, a state diagram, and an illustration of how this microcode is partitioned within the micro-op translation logic. Note two assembler directives: *micro_op_end* marks the end of a micro-op, and *decode* decodes any control vector directly from the opcode and is unique to μRTL microcode.

The first direct decode examples are the jump tables (*pg1_JTA* and *pg1_JTB*), which are used to link together different sequences of micro-ops depending on the opcode. This allows us to implement different addressing modes by jumping to the *LD_DIR_EXT:* or *LD_INDEXED: / LD_INDIRECT:* micro-ops first. However, in the case of immediate addressing, we jump directly to the *ADD:* micro-op since the data is already available as part of the instruction, resulting in single cycle execution. Note that the other addressing modes reach the *ADD:* micro-op via jump table B, having already used jump table A for the addressing mode. If this were a multi-cycle processor, we would jump to the fetch state after completing this execute state. However, since this is a pipelined design, the next instruction has already been fetched and decoded. Therefore, the *JUMP_TABLE_A_NEXT_PC* mnemonic directs the microsequencer to use the jump table A address decoded from the next instruction.

Direct decode logic is also used to consolidate micro-ops by identifying similar micro-ops that differ by only a few control vectors. An example is the *ADD:* micro-op which uses the *R1* and *R2* register pointers to implement all the different addressing mode and target register variations of the ADD instructions. The *pg1_R1* and *pg1_R2* decode tables define which registers *R1* and *R2* point to depending on the opcode. For example, the ADDA immediate instruction sets the *R1* pointer to the *A* register and the *R2* pointer is set to *IDATA* which is the instruction's immediate data. For the direct, indexed, indirect or extended addressing modes, *R2* points to the *DMEM_RD* register which is loaded by the previous micro-ops that implement the desired addressing mode.



Decoded R1 Register Pointer		Page 1 R1	Page 2 R1	Page 3 R1
A	A Accumulator	55		
B	B Accumulator	54		
PC	Program Counter	28	17	1
DMEM_RD	Data Memory Read	33		
D	D Accumulator	19	4	
X	X Index Register	14	2	
Y	Y Index Register	1	13	
U	U Stack Pointer	8		4
S	S Stack Pointer	1	7	4
CCR	Condition Code Reg	2		

µRTL Page 1 R1 Ptr Decode	opcode[7:4]															
	DIR		REL		ACCA	ACCB	IDX / IND	EXT	IMM	DIR	IDX / IND	EXT	IMM	DIR	IDX / IND	EXT
opcode[3:0]	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NEG		BRA	LEAX	NEGA	NEGB	NEG	NEG	SUBA	SUBA	SUBA	SUBA	SUBB	SUBB	SUBB	SUBB
1			BRN	LEAY					CMPA	CMPA	CMPA	CMPA	CMPB	CMPB	CMPB	CMPB
2			BHI	LEAS					SBCA	SBCA	SBCA	SBCA	SBCB	SBCB	SBCB	SBCB
3	COM		BLS	LEAU	COMA	COMB	COM	COM	SUBD	SUBD	SUBD	SUBD	ADDD	ADDD	ADDD	ADDD
4	LSR	EMUL	BHS,BCC		LSRA	LSRB	LSR	LSR	ANDA	ANDA	ANDA	ANDA	ANDB	ANDB	ANDB	ANDB
5		EMULS	BLO,BCS						BITA	BITA	BITA	BITA	BITB	BITB	BITB	BITB
6	ROR	LBRA	BNE		RORA	RORB	ROR	ROR	LDA	LDA	LDA	LDA	LDB	LDB	LDB	LDB
7	ASR	LBSR	BEQ		ASRA	ASRB	ASR	ASR		STA	STA	STA		STB	STB	STB
8	ASL,LSL	IDIV	BVC		ASLA,LSLA	ASLB,LSLB	ASL,LSL	ASL,LSL	EORA	EORA	EORA	EORA	EORB	EORB	EORB	EORB
9	ROL	DAA	BVS	RTS	ROLA	ROLB	ROL	ROL	ADCA	ADCA	ADCA	ADCA	ADCB	ADCB	ADCB	ADCB
A	DEC	ORCC	BPL	ABX	DECA	DECB	DEC	DEC	ORA	ORA	ORA	ORA	ORB	ORB	ORB	ORB
B			BMI	RTI					ADDA	ADDA	ADDA	ADDA	ADDB	ADDB	ADDB	ADDB
C	INC	ANDCC	BGE		INCA	INCB	INC	INC	CMPX	CMPX	CMPX	CMPX	LDD	LDD	LDD	LDD
D	TST	SEX	BLT	MUL	TSTA	TSTB	TST	TST	BSR	JSR	JSR	JSR		STD	STD	STD
E	JMP		BGT				JMP	JMP	LDX	LDX	LDX	LDX	LDU	LDU	LDU	LDU
F	CLR		BLE	SWI	CLRA	CLRB	CLR	CLR		STX	STX	STX		STU	STU	STU

Fig. 4. R1 Decode Table vs Opcode Encoding (1 of 15 μRTL Decode Tables)

Finally, Figure 4 visually highlights how the direct decode logic efficiently utilizes the existing opcode encoding of the Turbo9 / 6809 ISA, resulting in optimized logic synthesis.

IV. Analysis: μRTL vs Hardwired Decoder

This section will analyze the performance gain and efficiency advantage of μRTL micro-op translation over a hardwired decoder. The previous section described the implementation of the ADD instructions using the μRTL methodology. For direct comparison, we analyzed a different implementation of the same 6809 ADD instructions. The MC6809 IP core provides an excellent example of a hardwired instruction decoder [11]. This core is cycle-accurate with the original Motorola 6809 and is well-designed using Verilog, taking advantage of directly decoding control signals from the opcode.

TABLE I
μRTL Micro-Op States vs Hardwired States

Instruction	Addressing Mode	Turbo9 Micro-op States	MC6809 Hardwired States
ADD(AorB)	Immediate	1	2
ADD(AorB)	Direct	2	4
ADD(AorB)	Index/Indirect	2 to 3	4 to 12
ADD(AorB)	Extended	2	5
ADDD	Immediate	1	4
ADDD	Direct	2	6
ADDD	Index/Indirect	2 to 3	6 to 14
ADDD	Extended	2	7
States used per ADD instruction:		1 to 3	2 to 14
States required for all ADD instructions:		4	27
States required for entire instruction set:		66	93

The two implementations of the ADD instructions are compared in Table I. The MC6809 uses sequences of 2 to 14 hardwired states to process all variations of the ADD instructions, requiring a total of 27 different states. In contrast, the Turbo9 requires only 4 micro-op states to implement all variations of the ADD instructions and executes them in just 1 to 3 micro-ops. The μRTL microarchitecture also implements the entire 6809 ISA in 29% fewer states, even with the Turbo9's multiply and divide instruction set extensions. In conclusion, the μRTL methodology's pipelined implementation and reduced number of micro-op states are the main reasons for the impressive performance and area results presented in the next section.

V. Results: μRTL vs RISC Microarchitectures

We compared our μRTL microarchitecture to RISC microarchitectures, which are the common choice for deeply embedded microprocessors. These embedded IP cores matched the same criteria as the Turbo9: they must be optimized for area, have C compiler support, and be open-source. This allowed us to synthesize them into the same standard cell library and compare area and power consumption. Additionally, we compiled the popular C benchmark, Dhrystone version 2.1, to quantify integer performance.

The Atmel AVR is arguably the equivalent RISC 8/16-bit instruction set. The AVR Core has the same cycle timing as the popular ATmega series [12]. The core implements a combinational multiplier, and the compiler used was avr-gcc.

The RISC-V architecture is a common 32-bit embedded solution today, and the PicoRV32 core is a very popular area-optimized implementation [13]. We used the riscv32im-gcc compiler and the recommended Dhrystone setup for the core (fast_mul, div, and barrel_shifter enabled).

We employed two external memory bus configurations of the Turbo9: the Turbo9GTR, with separate 16-bit program and data memory buses, and the Turbo9S, with a simpler shared 16-bit memory bus. The vbcc compiler was used for both the Turbo9 and MC6809 [14].

Each compiler's optimal native integer precision (16-bit or 32-bit) is acceptable for deeply embedded applications, and we used the -O3 optimization settings.

A. Synthesis Results

Since the μRTL toolset and the Turbo9 are open-source, we chose the open-source OpenLANE ASIC design flow using the open-source Skywater Technology 130nm standard cell library to synthesize all IP cores [15]. All synthesis strategies were explored to optimize for area and delay. The area results were then normalized to 2-input NAND gate equivalence (kGE) and are shown in Table II.

TABLE II
OPENLANE & SKY130 SYNTHESIS RESULTS

Processor Core	**Area (kGE)**	**Delay (ns)**	**Internal Power (mW)**	**Switch Power (mW)**	**Leakage Power (nW)**	**Total Power (mW)**
PicoRV32	22.8	5.0	23.8	10.9	68.1	34.7
AVR Core	5.7	5.4	5.4	2.7	16.3	8.1
Turbo9GTR	5.3	3.6	4.9	2.5	16.9	7.4
Turbo9S	5.4	3.7	5.7	2.7	17.2	8.4
MC6809	6.9	3.5	5.3	4.0	17.6	9.3

Remarkably, the Turbo9's advanced pipelined microarchitecture is 21.8% smaller than the multi-cycle MC6809 core. This demonstrates that the μRTL methodology can produce efficient microarchitectures that are not only better in cycle by cycle performance, but also have smaller area and lower power. The Turbo9 is also smaller than both the PicoRV32 RISC-V and the AVR core by 76.3% and 5.3% respectively.

B. Performance Results

The Dhrystone performance is shown in Figure 5. The Turbo9GTR at 0.75 DMIPS/MHz is 44.2% faster than the PicoRV32 RISC-V, 82.9% faster than the AVR RISC, and 316.6% faster than the MC6809. The Turbo9S with its limited 16-bit memory bus bandwidth delivers impressive results with 0.68 DMIPS/MHz and still far outperforms the AVR with its 24-bits of memory bandwidth, and the PicoRV32 with its 32-bits of memory bandwidth.

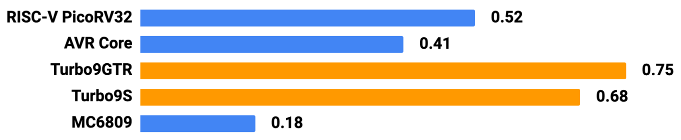


Fig. 5. Performance Per Clock (DMIPS / MHz, higher is better)

Performance relative to die area is more important for embedded microarchitectures than absolute performance. The performance vs area efficiency is shown in Figure 6 given a clock rate of 100MHz. Both RISC architectures with numerous internal registers fall further behind the Turbo9's efficient combination of an accumulator ISA with μRTL micro-op translation making it preferred for embedded applications.

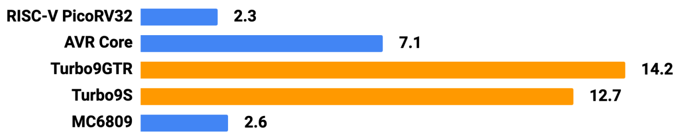


Fig. 6. Performance at 100MHz vs Area (DMIPS / kGE, higher is better)

VI. CONCLUSION

By looking beyond the accepted embedded RISC solutions for more compact architectures targeting area-constrained SoC sub-blocks, we developed the μRTL toolset. With this new tool, we utilized the high-performance micro-op translation techniques common in large super-scalar processors, allowing us to create a pipelined implementation of an accumulator-based instruction set. The resulting innovative microarchitecture delivers high performance in a small footprint, addressing the demand for space-efficient yet powerful embedded solutions. Given our encouraging initial results, we are optimistic about several future opportunities to further explore the potential of the μRTL methodology and Turbo9 IP for compact, high-performance embedded applications where size and efficiency matter.

REFERENCES

[1] K. Saso and Y. Hara-Azumi, "Revisiting Simple and Energy Efficient Embedded Processor Designs Toward the Edge Computing," *IEEE Embedded Systems Letters*, Jun. 2020.
[2] B. Fu, A. Saini, and P. Gelsinger, "Performance and microarchitecture of the i486 processor," in *Proceedings 1989 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, Oct. 1989.
[3] J. P. Shen and M. H. Lipasti, *Modern Processor Design: Fundamentals of Superscalar Processors*. McGraw-Hill, 2005.
[4] G. G. Henry, "The VIA Isaiah Architecture," *Centaur Technology*, 2008.
[5] Patterson and Hennessy, *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. Morgan Kaufmann, 2017.
[6] M. A. Lynch, *Microprogrammed State Machine Design*. Boca Raton, FL: CRC Press, Jan. 1993.
[7] K. Phillipson and M. Rywalt, "Turbo9 - A Compact & Efficient Pipelined 6809 Microprocessor IP," May 2024. [Online]. Available: https://www.turbo9.org
[8] K. Phillipson, "A Compact and Efficient Microprocessor IP for SoC Sub-Blocks and Mixed-Signal ASICs," Master's thesis, University of Florida, 2022. [Online]. Available: https://ufdcimages.uflib.ufl.edu/UF/E0/05/87/40/00001/Phillipson_K.pdf
[9] B. Pitre and M. Margala, "A Novel Approach to Managing System-on-Chip Sub-Blocks Using a 16-Bit Real-Time Operating System," *MDPI*, Jan. 2024. [Online]. Available: https://www.mdpi.com/2079-9292/13/10/1978
[10] T. Ritter and J. Booney, "A Microprocessor for the Revolution: The 6809," *BYTE Magazine*, Jan. 1979.
[11] G. Miller, "Cycle Accurate MC6809 Core," Sep. 2023. [Online]. Available: https://github.com/cavnex/mc6809
[12] R. Lepetenok, "Overview :: AVR Core :: OpenCores," Feb. 2017. [Online]. Available: https://opencores.org/projects/avr_core
[13] C. Wolf, "PicoRV32 - A Size-Optimized RISC-V CPU," Aug. 2023. [Online]. Available: https://github.com/YosysHQ/picorv32
[14] V. Barthelmann, "Advanced compiling techniques to reduce RAM usage of static operating systems," Ph.D. dissertation, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), 2004.
[15] M. Shalan and T. Edwards, "Building OpenLANE: A 130nm OpenROAD-based Tapeout- Proven Flow : Invited Paper," in *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, Nov. 2020.