A COMPACT AND EFFICIENT MICROPROCESSOR IP FOR SOC SUB-BLOCKS AND MIXED-SIGNAL ASICS

By

KEVIN PHILLIPSON

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2022

To my wife, Callie, and our children, Wyatt and Eleanor

ACKNOWLEDGMENTS

During the research and writing of this thesis, I have received amazing support from friends and family. First, I would like to thank my wife for her constant encouragement, patience, and love. Without her support, I would not have been able to accomplish this work while managing a full-time job and raising a family. Next, I would like to thank one of my best friends, Michael Rywalt, who worked alongside me on this project. Michael wrote the custom microcode assembler, μRTL, and much of the verification testbench. But most importantly I am grateful for the time we spent working closely together to advance the project. Additionally, this project has brought me a new friend, Boisy Pitre, who assisted the project with compiler selection and setup some of the benchmarks. The enthusiasm he brought to the project is appreciated.

From the academic side I express my gratitude to University of Florida's Electrical and Computer Engineering department. I would like to thank my committee chair Dr. Greg Stitt, whose experience and guidance has been vital. I also want to recognize Dr. Herman Lam and Dr. William Eisenstadt for their invaluable contributions. I want to highlight not only Dr. Eric Schwartz's role in my academic career, but his impact on my life as a mentor. Additionally, I wish to recognize the influence of the late Dr. Michel A. Lynch, who ignited my passion for computer architecture and microprocessor design. Finally, I would like to acknowledge some of the other professors whose efforts helped shape my career: Dr. John Harris, Dr. Robert Fox & Dr. Martin Margala.

Furthermore, I would like to thank the following people. Terry Ritter and Joel Boney for their excellent work designing and documenting the Motorola 6809 instruction set. Dr. Volker Barthelmann for his outstanding C compiler, vbcc. My co-workers through years that have worked by my side and especially those that have served as mentors including Gordon Franza.

My engineering friends who always encourage my pursuits, including: Aaron Tucker, Brian Pietrodangelo, Brad Atherton, and Mike Evans. Last, I would like to thank my mother and father for their unwavering support and my brother who will always be my best friend.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ADD	6809 addition instruction
ALU	Arithmetic Logic Unit
AMBA	ARM's Advanced Microcontroller Bus Architecture
APB	ARM's Advanced Peripheral Bus
ARM	A commercial family of RISC architectures
ASIC	Application Specific Integrated Circuit
CCR	6809's Condition Code Register
CISC	Complex Instruction Set Computer
CMOS	Complementary Metal Oxide Semiconductor
CPU	Central Processing Unit
DDR	Double Data Rate memory interface
DMIPS	Dhrystone MIPS
DSP	Digital Signal Processor
DV	Digital Verification
EA	6809's Effective Address
FIFO	First In First Out buffer
FPGA	Field Programable Gate Array
GCC	GNU C Compiler
GPU	Graphics Processing Unit
I2C	Inter-Integrated Circuit bus
IC	Integrated Circuit
IP	Intellectual Property
ISA	Instruction Set Architecture

LUT	Logic Look Up Table
MIPS	Million Instructions Per Second
NAND	AND logic gate with inverted output
NIC	Network Interface Controller
PCB	Printed Circuit Board
PCIe	Peripheral Component Interconnect Express
RISC	Reduced Instruction Set Computer
RISC-V	A open-source family of RISC architectures
ROM	Read Only Memory
RTL	Register Transfer Language
SAU	Sequential Arithmetic Unit
SoC	System on Chip
SPI	Serial Peripheral Interface
UART	Universal Asynchronous Receiver Transmitter
uRTL	Turbo9's microcode to RTL assembler
USB	Universal Serial Bus
vbcc	Volker Barthelmann's C Compiler

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

A COMPACT AND EFFICIENT MICROPROCESSOR IP FOR SOC SUB-BLOCKS AND MIXED-SIGNAL ASICS

By

Kevin Phillipson

May 2022

Chair: Greg Stitt
Major: Electrical and Computer Engineering

System on Chip (SoC) design is a fundamental part of building modern electronic devices. SoCs consist of many levels of hierarchy; most apparent are the large multi-core CPU and GPU blocks. However, a significant portion of the SoC is comprised of sub-blocks such as memory interfaces, storage, network, and human interfaces. These sub-blocks often require a compact microprocessor for reprogrammable high-level control. Suitable tasks include design for test circuitry, high-level control of analog or DSP blocks, or management of an interface such as SPI, USB, etc.

Most of the professional microcontroller IP cores available are scaled down versions of 32-bit architectures, but many of these tasks only require 16-bit precision. From our experience there is a demand for a small and efficient 8/16-bit microprocessor. Most of the 16-bit microprocessor IPs currently available are hobbyist projects that lack clean structured RTL, area/power/performance synthesis optimization, a full self-checking testbench, and/or C compiler support.

In this thesis we introduce the Turbo9, an open-source professional microprocessor IP for use in SoC sub-blocks or mixed signal ASICs. The Turbo9 is a new pipelined microarchitecture that executes the Motorola 6809 instruction set. The accumulator-based ISA of the 6809 was

chosen because of its simplicity, elegance, and compatibility with C compilers. Its 16-bit support and excellent addressing modes map nicely to stack-relative addressing and pointers. Using benchmarks such as Dhrystone, we have shown the performance of the Turbo9 to be equal to or better than the 32-bit architectures while consuming less area and power.

CHAPTER 1
INTRODUCTION

Modern electronic devices have seen a constant trend of increasing integration. The discrete components and integrated circuits that once occupied a large printed circuit board (PCB) are now being consolidated into a single chip known as a System on Chip (SoC). At the PCB level there appears to be a reduction of complexity. However, looking inside the SoC, the levels of hierarchy and overall complexity have greatly increased. This high level of integration has allowed for greater functionality and capability of modern electronic devices.

We see the need for a compact and efficient microprocessor IP for use in these modern SoCs. To fulfill this need, we developed the Turbo9 microprocessor [1]. The Turbo9 is an open-source compact high performance pipelined 16-bit microprocessor IP, delivered in a professional package. We can show that it is the superior compact microprocessor solution for certain SoC applications versus other commonly used 32-bit solutions.

Motivation

A typical SoC contains a large application processor as well as many sub-blocks. Examples of these sub-blocks include:

- Digital Signal Processing Blocks
- Analog / Mixed-Signal Blocks
- High Speed Storage Interfaces
    - PCIe
    - DDR
- Communication Interfaces
    - Ethernet / Wireless
    - USB
    - SPI / I2C / UART
- Human Interfaces
    - Touch or Keyboard / Mouse
    - Video
    - Audio

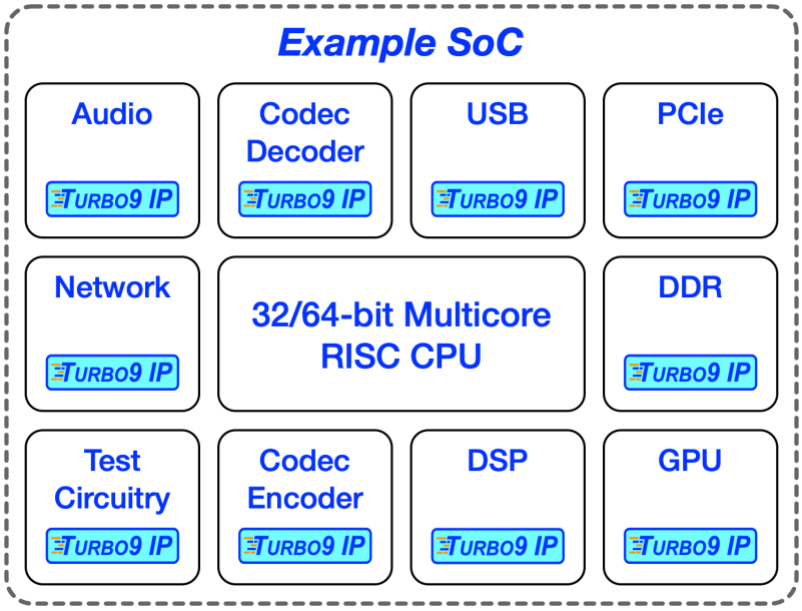


Figure 1-1. Example top level block diagram of a SoC.

Our target applications are these SoC sub-blocks that need a compact microprocessor for programmable high-level control. Also, outside of large SoCs there are many smaller mixed-signal ASICs used for sensors and other applications that could also use a compact and efficient microprocessor IP.

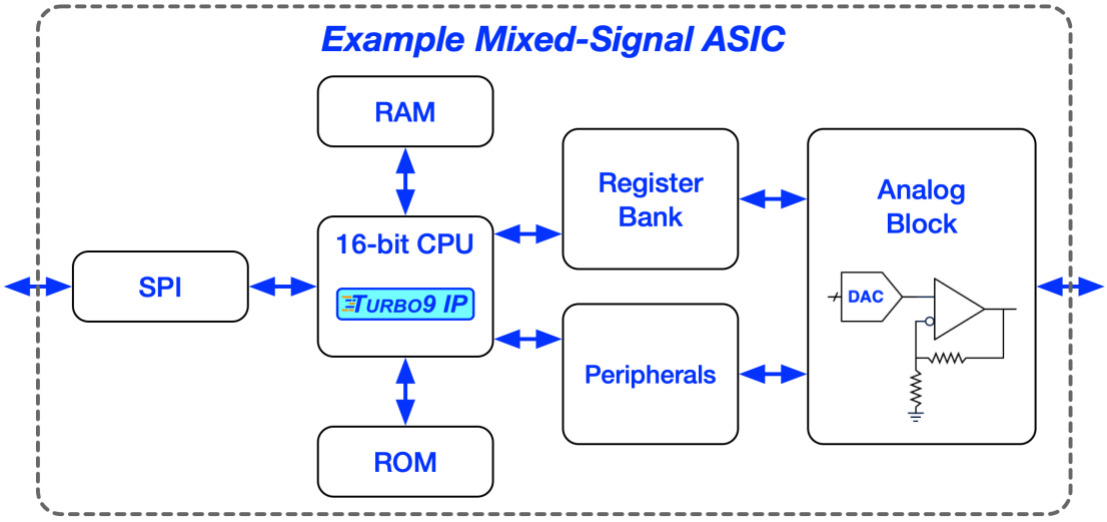


Figure 1-2. Example top level block diagram of a mixed-signal ASIC

We want to fill a niche between a synthesized state machine and a 32-bit RISC microcontroller. We believe our IP will be useful for the following tasks:

- Design for test circuitry
- High-level control of analog or DSP blocks
- Managing an interface (SPI, USB, NIC, etc.)

For the above tasks, we assert that a compact and efficient 16-bit processor is the correct solution. The use-case comparison shown in Table 1-1 summarizes our reasons for choosing a 16-bit architecture over a 32-bit architecture.

Table 1-1. Microprocessor 16-bit vs 32-bit comparison

	16-bit CPU use case	32-bit CPU use case
Typical Application	Microcontrollers	Application processors
Die Area	Small die area	Large die area
Power	Lower given same clock frequency	Higher given same clock frequency
Memory Usage	Small internal memory	Large external memory
16-bit Performance	Equal	Equal
32-bit Performance	Worse	Better

Generally speaking, if the task only requires 16-bit precision, a larger 32-bit CPU can be a waste of area. Moreover, given the same clock frequency, the 32-bit CPU will also use more power. Given this reasoning, we believe there is a demand for a compact, high performance 16-bit CPU with efficient memory utilization.

Instruction Set Architecture Choice

Given that we have determined a 16-bit microprocessor would be the best fit for these requirements, we must select either an existing instruction set architecture or design our own. If we select an existing instruction set, we can take advantage of third-party tools, compilers, and assemblers, rather than having to also develop those tools. We chose the Motorola 6809 instruction set because we believe it has the minimum feature set to support a C compiler well. Figure 1-3 shows the programming model of the 6809. It is an accumulator architecture but

would be considered CISC under the retroactive definition given by the RISC philosophy [2]. The 6809 is not a load-store architecture and this violates the RISC definition. It is an accumulator architecture which means its instructions use memory as one of the operands. But the instruction set itself is actually much more compact than most RISC instruction sets. Also, the instruction set is very orthogonal which can be seen in the opcode matrix in Figure A-1.

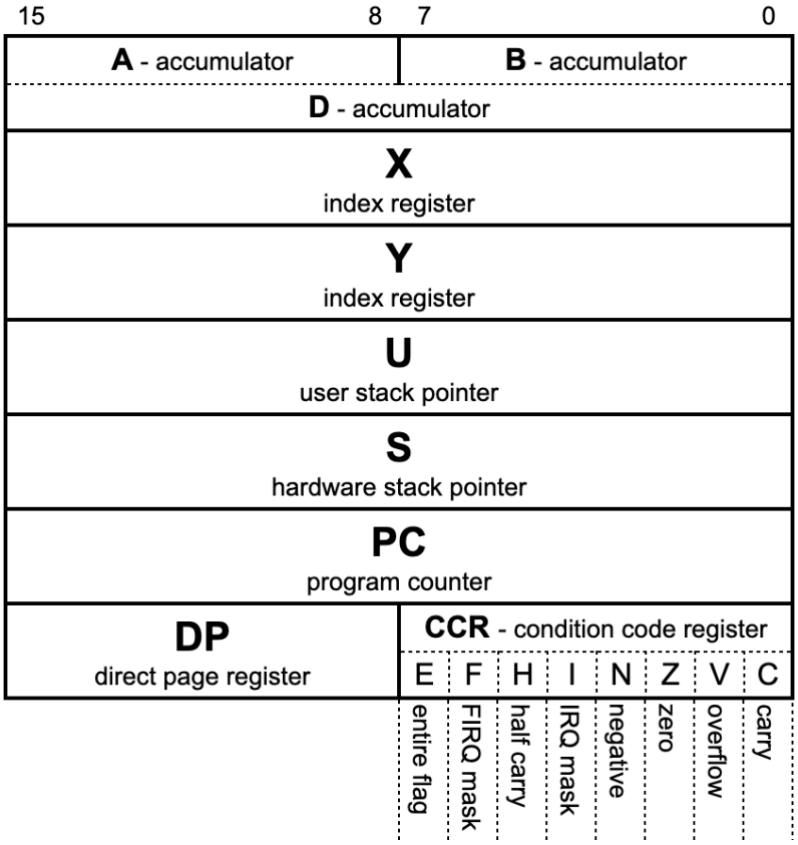


Figure 1-3. Motorola 6809 programming model showing internal registers

Despite its simplicity, the 6809 instruction set has several powerful features which make it very attractive [3]. Even though the original microarchitecture built by Motorola in 1978 has 8-bit data paths, the instruction set supports 16-bit data directly. Therefore, our Turbo9 microarchitecture can be a full 16-bit design to take advantage of these instructions. Second, the 6809 uses variable length instructions which are highly encoded and should result in better code density than loosely encoded RISC instructions. Finally, the addressing modes, and in particular

the powerful indexed and indirect addressing modes, map well to C concepts, such as arrays and pointers.

The choice of this instruction set might seem to buck the trend of RISC based architectures, but upon closer examination of the 6809 ISA, there is a good opportunity to create a small powerful IP that also needs less program memory. The reader should seek out documentation of the 6809 instruction set [4], if necessary, because this thesis assumes prior knowledge of that ISA.

Open-Source and Professional IP

Our objective is to design an open-source professional-level microprocessor IP. Most open-source microprocessor IPs are hobbyist projects, and few open-source IPs reach the professional level of RISC-V or ARM. To guarantee a professional-level and a high-quality product, certain standards and specifications must be selected and adhered to. First, because we have chosen an established ISA, we can exploit the third-party software tools, assemblers, compilers and the presence of an existing code base. It is also very important to use an industry standard SoC bus for ease of integration into the design.

We approach the design and optimization of the microprocessor using professional tools and techniques. Modern RTL design practices should be used. The design should be fully synchronous with a single clock. There should be a well-defined separation of the control and data paths and an effort to break the design into smaller modules that are easier to maintain. The design optimization should be driven by feedback from synthesis into ASIC standard cell libraries and FPGAs. This optimization should be a balance of speed, power, and area. This is achieved by using advanced microarchitecture design for performance, but not at the expense of power and area. For example, timing paths should be minimized for the maximum clock rate, but the tradeoffs of pipelining vs. multicycle methodologies should always be considered.

Finally, a professional design must be verified by means of a Design Verification (DV) test bench. A list of verification attributes should be developed that covers every possible instruction and addressing mode combination in the ISA. The Verilog testbench should then use directed and randomized tests to cover all these attributes. Once this testbench is ready, we will run a nightly regression that constantly checks the functionality during our optimization and development to ensure that the design still meets specifications.

CHAPTER 2
TURBO9 MICROARCHITECTURE OVERVIEW

As discussed previously, the Turbo9 is a 16-bit pipelined microarchitecture that executes the Motorola 6809 instruction set. Figures 2-1 and 2-2 show simplified and detailed block diagrams with the Turbo9 connected to an external memory.

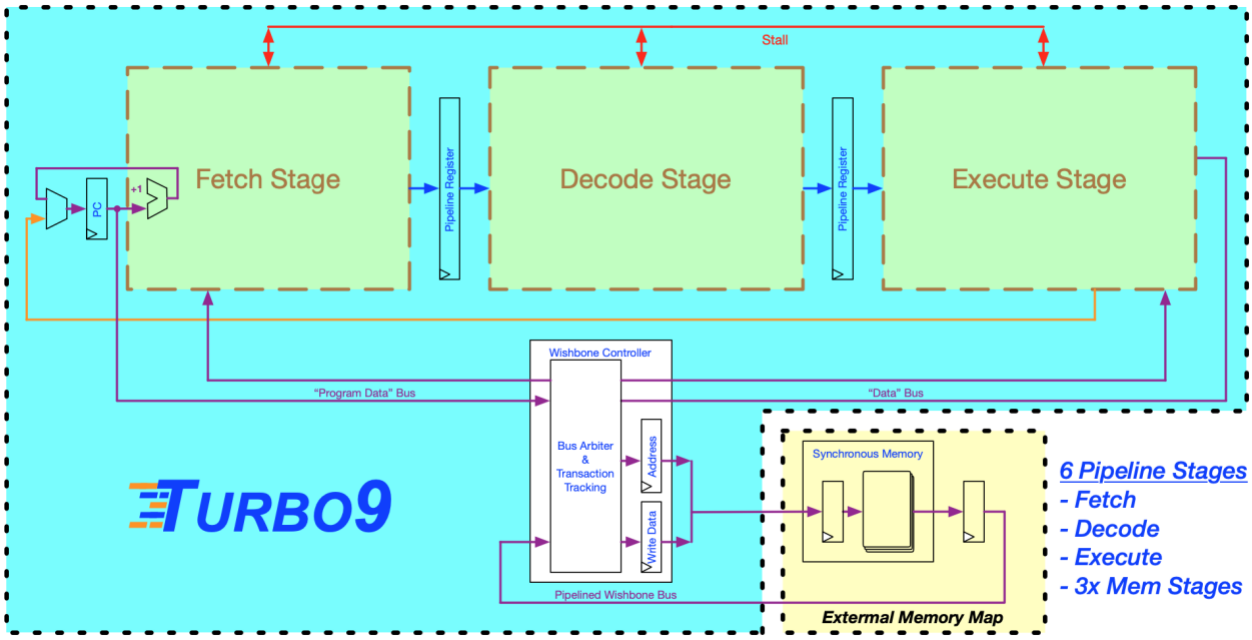


Figure 2-1. Top level block diagram of Turbo9 microarchitecture

Referring to Figure 2-1, the Turbo9's pipeline consists of six stages, three of which are the fetch, decode and execute stage. In this example, the three remaining stages are comprised of the Wishbone [5] controller and pipeline stages in the External Memory Map. Also, note that external memory bus of the Turbo9 is a Von Neuman (shared data/program) memory interface, but internally it has a Harvard architecture. There is a separate program memory bus connected to the fetch stage and a data memory bus connected to the execute stage.

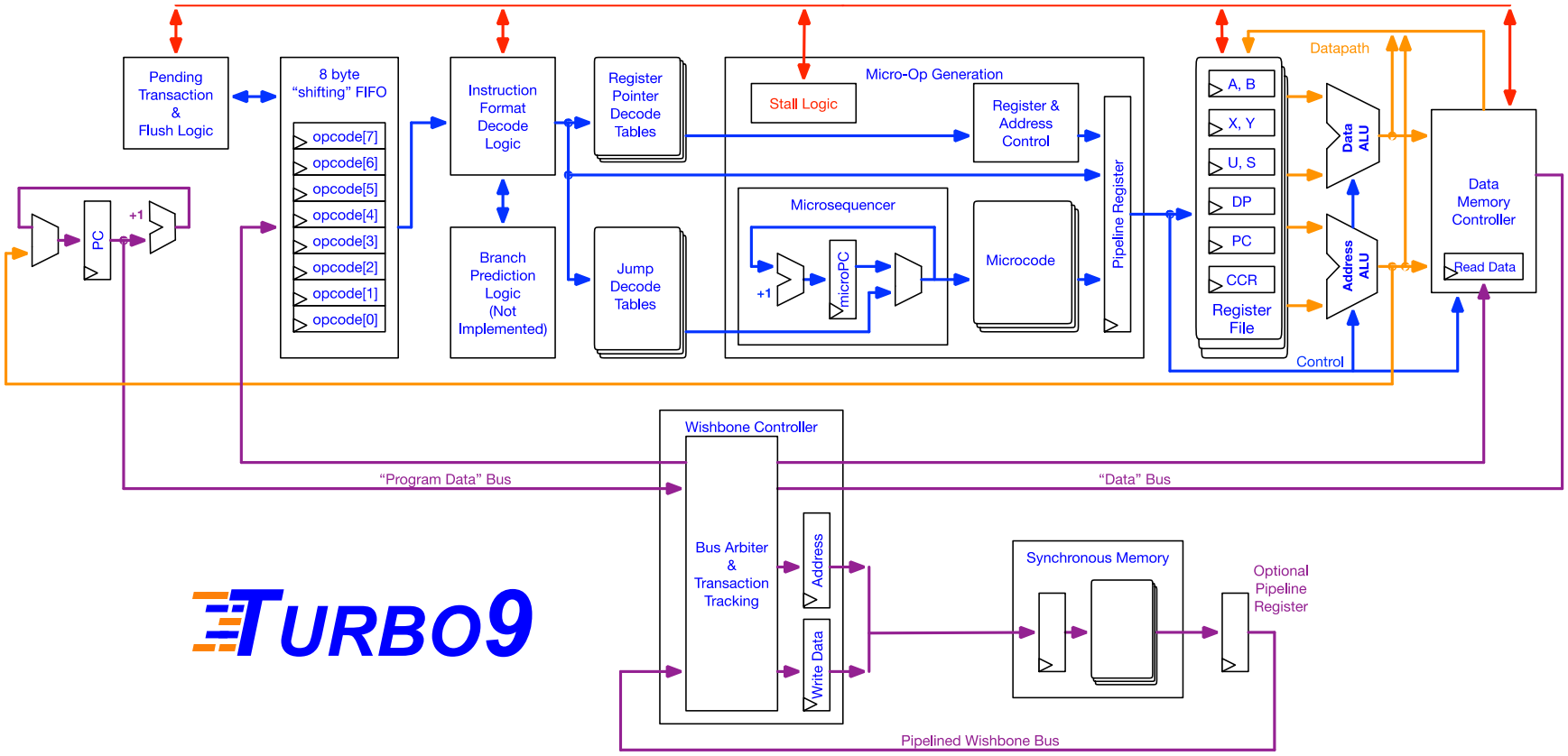


Figure 2-2. Detailed top level block diagram of Turbo9 microarchitecture

Pipelined Memory Bus Architecture

In order to meet the standard of professional level SoC IP, the Turbo9 needs to implement an industry standard memory bus interface. Another one of our requirements was a pipelined memory architecture to take advantage of synchronous memories common in modern ASICs. Additionally, the chosen bus standard needs to be available with a permissive open-source license since the Turbo9 is an open-source IP. The last concern is implementation requirements. We want the bus interface block in the Turbo9 to be modular and easily replaceable with a different SoC bus standard such as ARM's AMBA (APB) specification [6].

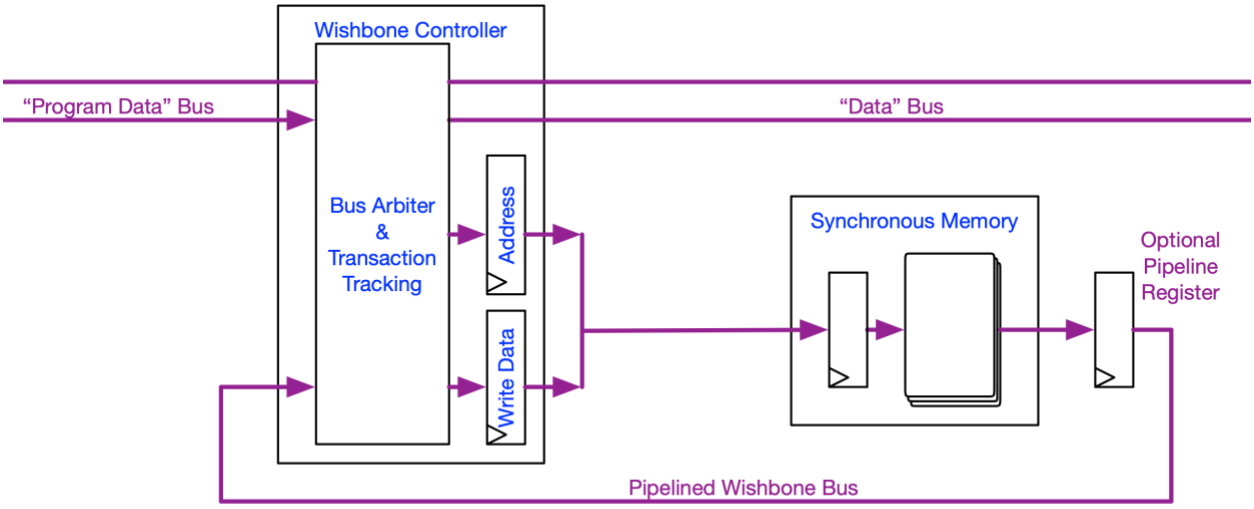


Figure 2-3. Turbo9's pipelined Wishbone SoC bus

Pipelined Wishbone SoC Bus

Given our stated requirements, we chose the OpenCores Wishbone SoC bus [5], specifically the Wishbone B4 pipelined version. This standard is simple to implement and integrates well with our pipelined microarchitecture. It is completely public domain, which does not restrict the open source nature of the Turbo9 IP. Figures 2-4 and 2-5 show a typical read and write cycle.

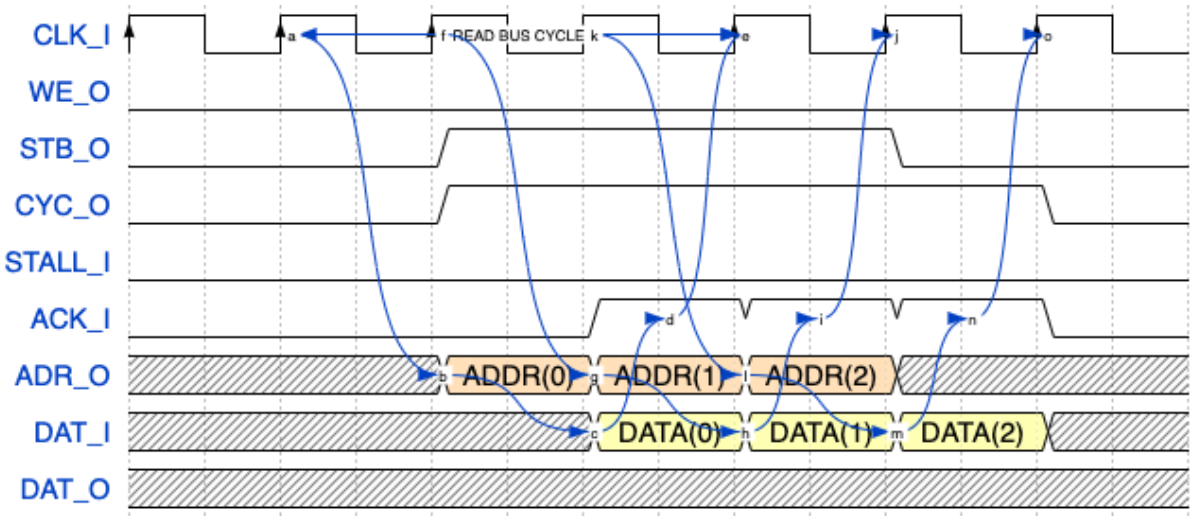


Figure 2-4. Pipelined Wishbone bus read cycle (3 cycle latency, 1 cycle throughput)

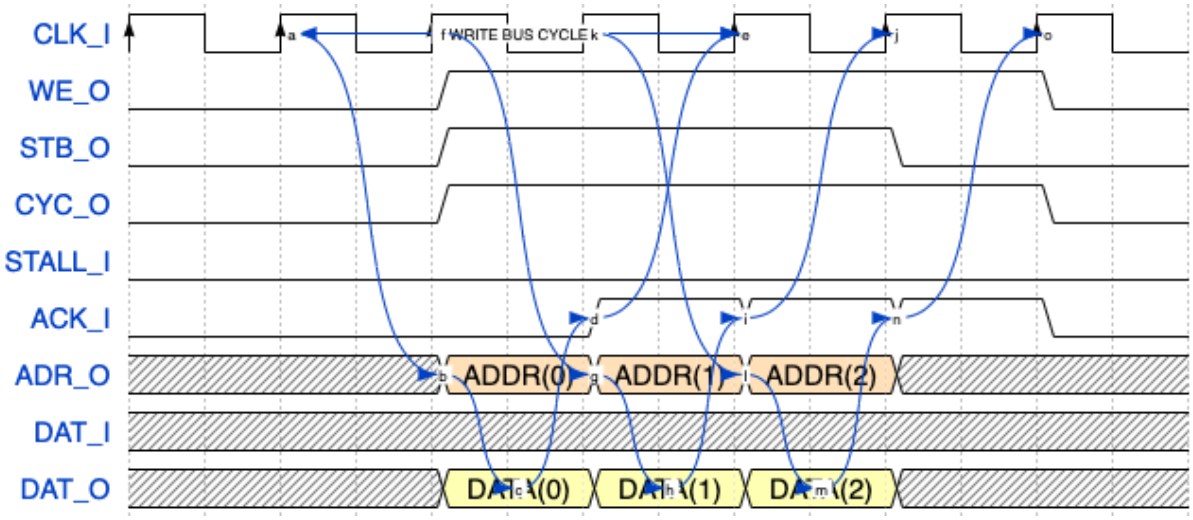


Figure 2-5. Pipelined Wishbone bus write cycle (3 cycle latency, 1 cycle throughput)

Configurable 16-bit or 8-bit Bus

Because the Wishbone interface of the Turbo9 is modular, we also developed three different data bus variations shown in Table 2-1. The standard variation uses an 8-bit data bus width, but we have also developed two versions with 16-bit buses. The 6809 instruction set architecture specifies 8-bit and 16-bit data can be stored on byte boundaries. Therefore the 16-bit versions of the Turbo9 can move 16-bit data faster over their wider bus. The two versions,

Turbo9S and Turbo9R, differ in their ability to handle non-aligned 16-bit words. The Turbo9S takes one cycle to read/write an aligned 16-bit word and two cycles for an unaligned 16-bit word. The Turbo9R can handle aligned and unaligned 16-bit transactions in one cycle.

Table 2-1. Turbo9 variations

	Internal data-path	External data-bus	8-bit throughput	16-bit throughput
Turbo9	16-bit	8-bit	1 cycle	2 cycles
Turbo9S	16-bit	16-bit	1 cycle	1 cycle aligned / 2 cycles unaligned
Turbo9R	16-bit	16-bit	1 cycle	1 cycle

Automatic Latency Adjustment

An additional feature we provide to the SoC designer is automatic latency adjustment. We have gone to great effort to make the Turbo9 itself as efficient as possible, but the SoC designer is responsible for connecting the external peripherals and memories. We have also gone to great effort to give the designer the tools necessary to create an efficient memory subsystem.

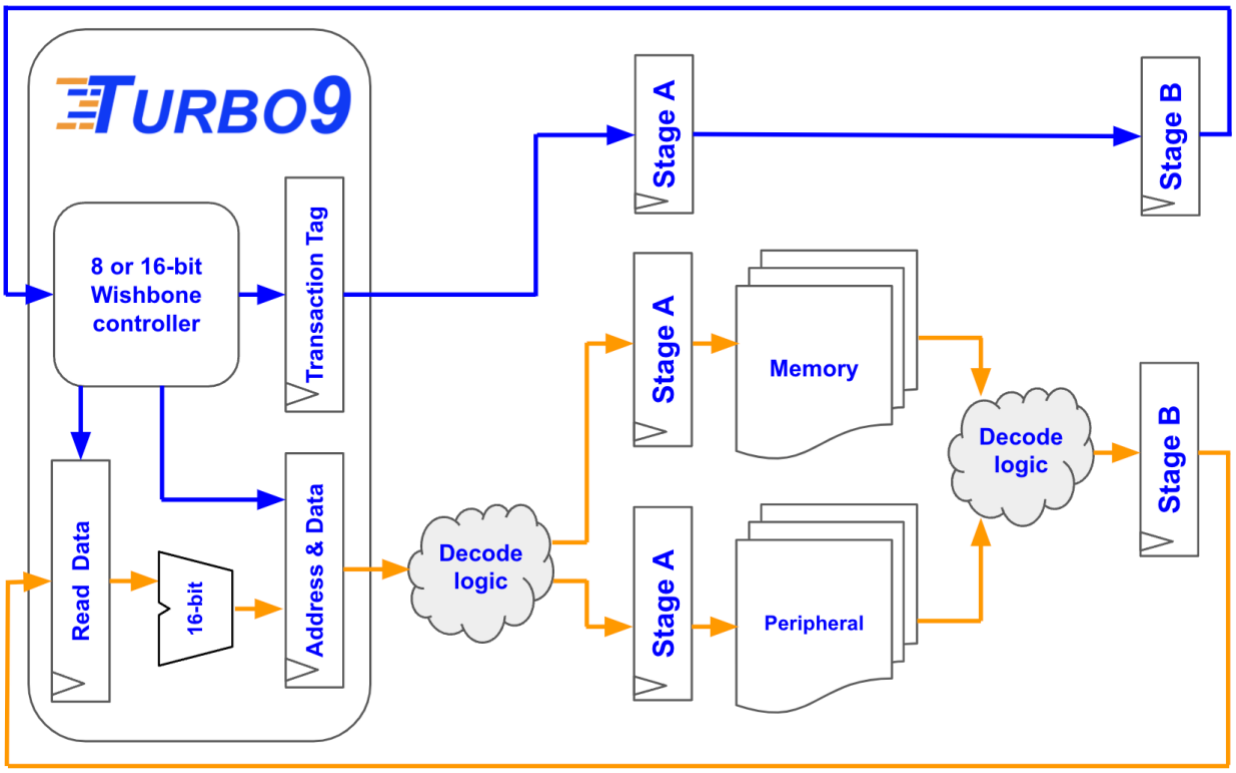


Figure 2-6. Turbo9's pipelined Wishbone SoC bus typical implementation

Figure 2-6 shows a typical Turbo9 with a typical memory subsystem. Note the pipeline registers Address/Data/Tag, Stage A, Stage B, and Read Data. Given this configuration, the latency will be four clock cycles, but any one of these register stages can be removed or run on the opposite clock edge, and the Turbo9's Wishbone controller is intelligent enough to automatically adjust.

Of course, the lowest latency will improve performance if maximum frequency is not considered. But the memory subsystem could become the worst-case timing path and limit the maximum frequency of the entire design. In this case, adding extra pipeline stages is desirable. The designer must make the following tradeoff:

- More pipeline stages
  - Higher latency
  - Higher maximum frequency
  - Lower efficiency
- Less pipeline stages
  - Lower latency
  - Lower maximum frequency
  - Higher efficiency

The Turbo9's 8-bit and 16-bit variations of its Wishbone controller, along with its automatic latency adjustment gives the SoC designer the tools to make an efficient and high-performance memory subsystem.

Fetch Stage

The fetch stage (Figure 2-7) fetches program data from the Wishbone controller and provides a complete instruction to the decode stage. It consists of an eight byte instruction queue, its own program counter, and pending transaction / flush logic. It interfaces with the Wishbone controller and reads program data whenever the data bus does not have priority. It also has its own speculative program counter (PC) that can be reloaded from the "true" PC in the execute stage when an instruction queue flush is requested.

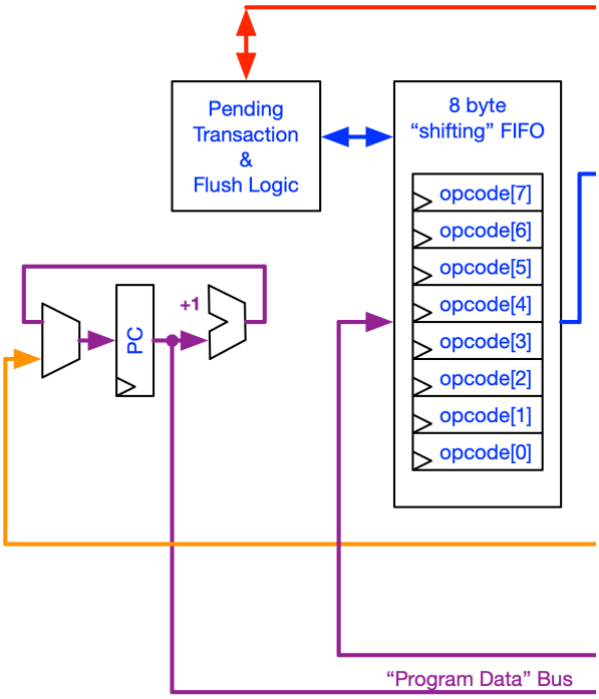


Figure 2-7. Turbo9's fetch stage

Instruction Queue

The fetch stage contains an 8-byte instruction queue. This is not an instruction cache but does do some level of predictive instruction loading. Unless interrupted by the execute stage, the fetch stage is constantly loading program data into the instruction queue. Also note, the instruction queue is used as a pipeline register between the fetch and decode stages. Because of this, a shifting FIFO was used to implement the instruction queue rather than a circular FIFO. This was necessary to keep all combinatorial logic within the fetch stage and present only registered outputs to the decode stage.

Pending Transaction & Flush Logic

Due to the latency of the memory subsystem, the fetch stage includes pending transaction tracking logic so it can keep a record of read requests. When read data comes back, it can place the data into the instruction queue. This is also important in case a flush request is sent from the execute stage. If there is an incorrect branch prediction, it is important to know how many

pending transactions there are on the Wishbone bus so that reads coming back can be ignored. Of course, the instruction queue itself would also be emptied on a flush request.

Decode Stage

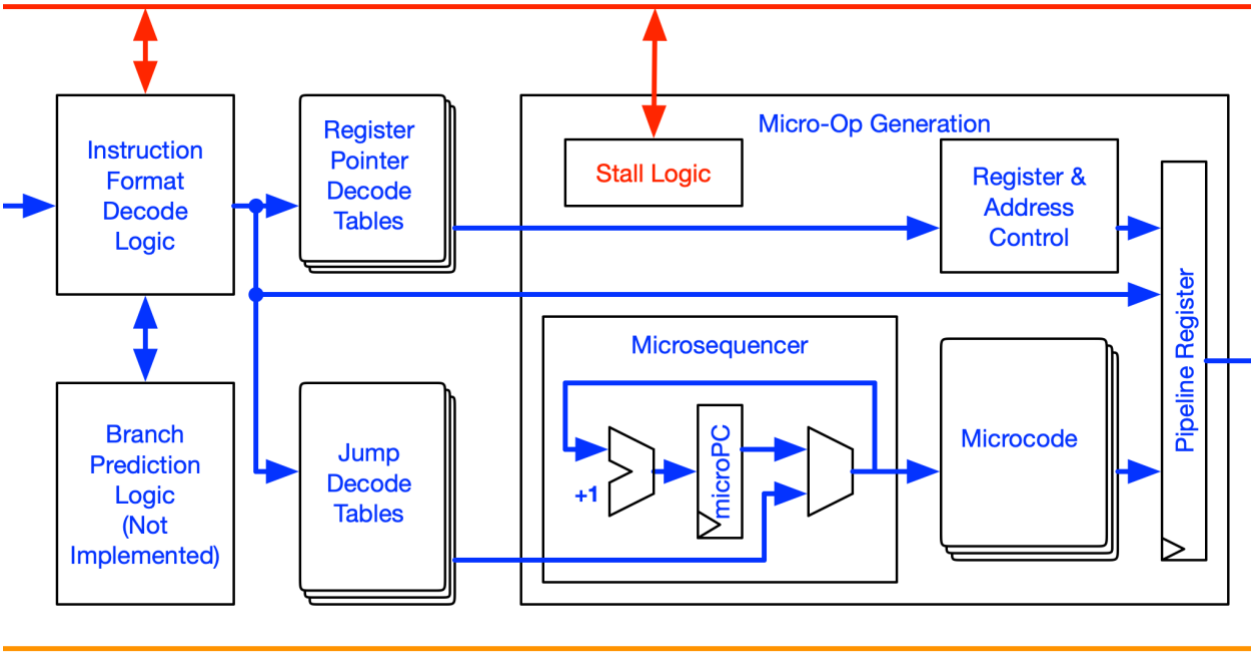


Figure 2-8. Turbo9's decode stage

The decode stage (Figure 2-8) breaks down 6809 instructions into RISC like micro-ops that are then delivered to the execute stage. Some instructions can be executed with a single micro-op; for example instructions that use inherent addressing (only internal data) or immediate addressing (data contained within the instruction). Instructions with indexed, extended, direct, indirect and relative address typically require one to three micro-ops. The decode stage must be able to produce the sequences of these micro-ops efficiently and has arguably the most influence on the area, power, and performance of the processor micro-architecture. To construct the Turbo9's decode stage we created the µRTL methodology which employs microcode abstraction techniques and applies them to a synthesized hardwired decode stage. The in-depth details are presented in Chapter 3.

Execute Stage

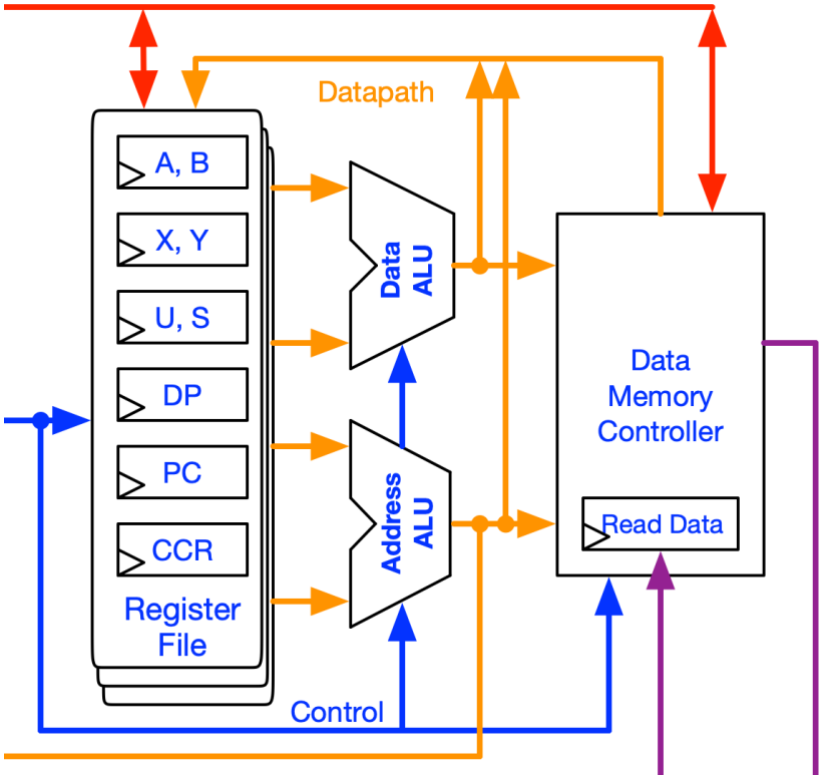


Figure 2-9. Turbo9's execute stage

The execute stage (Figure 2-9) contains all the major arithmetic logic, data paths, and registers in the 6809 programming model. Because the arithmetic functions and these data paths can often become the worst timing path, a great effort was made to ensure the execute stage was as efficient and elegant as possible. Many of the corner cases during the implementation of the 6809 instruction set were pushed back into the decode stage, and the execute stage just has to concern itself with simple RISC-like micro-ops.

Register File

The register file contains all the registers visible to the programmer in the 6809 ISA. It also contains a register called Effective Address (EA) that is not visible to the programmer. In addition, it contains all the decode logic for the register write-enables and the multiplexers to feed the data and address ALU.

Data ALU

The original Motorola 6809 implementation had an 8-bit ALU even though the instruction set includes 16-bit operations. The Turbo9 implementation has a fully 16-bit data ALU to exploit these 16-bit instructions for maximum performance. Figure 2-10 shows the data ALU's operation control vector. The first seven control vector symbols define the common arithmetic and logic functions found in a typical ALU. All functions can be performed on 8-bit or 16-bit data depending on another control vector called Data ALU Width Select.

```
; ////////////////////////////////////////// cv_DATA_ALU_OP definition
ctrl_vec_begin cv_DATA_ALU_OP 3
  A_PLUS_B      EQU $0 ; Y = A + B
  A_PLUS_NOT_B  EQU $1 ; Y = A + ~B
  LSHIFT_A      EQU $2 ; Y = A << 1
  RSHIFT_A      EQU $3 ; Y = A >> 1
  A_AND_B       EQU $4 ; Y = A & B
  A_OR_B        EQU $5 ; Y = A | B
  A_XOR_B       EQU $6 ; Y = A ^ B
  SAU           EQU $7 ; Y = SAU_Y
ctrl_vec_end
; //////////////////////////////////////////
```

Figure 2-10. Turbo9 data ALU operation control vector µRTL microcode definition

Sequential Arithmetic Unit

The last control vector symbol, "SAU", shown in Figure 2-10, is used to insert the Sequential Arithmetic Unit into the data path. The Sequential Arithmetic Unit is used to implement the 6809's multiply instruction as well as new multiply and divide instructions from the Motorola 68HC11 and 68HC12 ISAs.

New multiply and divide instructions derived from the HC11 and HC12 ISAs are useful for DSP and control applications. These instructions are parameterized in the Verilog RTL so the microprocessors instruction set extensions can be customized by the designer. The designer has the option to add instructions for performance or remove them to save area. Table 2-2 gives a summary of the instructions implemented in the SAU. The Sequential Arithmetic Unit is

designed with registers on its inputs and outputs. This encapsulates the logic and does not add to the worst-case propagation path in the data ALU.

Table 2-2. Sequential Arithmetic Unit: instructions implemented

6809 Instructions	
MUL	8 by 8 unsigned multiply
DAA	BCD correction
New 68HC11 Instructions	
IDIV	16 by 16 unsigned int divide
FDIV	16 by 16 unsigned frac divide
New 68HC12 Instructions	
EMUL	16 by 16 unsigned multiply
EMULS	16 by 16 signed multiply
EDIV	32 by 16 unsigned int divide
EDIVS	32 by 16 signed int divide
IDIVS	16 by 16 signed int divide
Possible Future Instructions	
COS	CORDIC cosine
SIN	CORDIC sine
SQRT	CORDIC square root

Address ALU

The Turbo9 also utilizes a 16-bit Address ALU in parallel with the data ALU which allows for concurrent effective address calculations during a data operation. One of the most powerful features of the 6809 ISA is its indexing addressing modes. The address ALU and the associated logic in the decode stage are optimized to calculate the index address in one micro-cycle and the indirect address in two micro-cycles.

Data Memory Controller

The Data Memory Controller manages the interface between the execute stage's data path and the Wishbone SoC bus controller. Figure 2-11 shows the Data Memory operation control vector definition. These operations can be 8-bit or 16-bit by setting the Data Width control vector appropriately. In some regard, the Data Memory Controller can be considered another pipeline

stage. Once a micro-op requests a read or write, the Data Memory Controller takes over and manages the transaction while the execute stage can proceed to the next micro-op.

```
; ////////////////////////////////////// cv_DMEM_OP definition
ctrl_vec_begin cv_DMEM_OP 2
  DMEM_OP_IDLE      EQU $0 ; Idle
  DMEM_OP_RD        EQU $2 ; Read Data
  DMEM_OP_WR        EQU $3 ; Write Data
ctrl_vec_end
; //////////////////////////////////////
```

Figure 2-11. Turbo9 data memory operation control vector uRTL microcode definition

CHAPTER 3
CISC TO RISC MICRO-OP PIPELINE DECODE STAGE

The Turbo9 utilizes a modern and customized decode stage that converts the 6809 CISC instructions into fixed length RISC micro-operations or “micro-ops.” This CISC-to-RISC translation layer is common in many modern high-performance CISC designs such as the x86-64 implementations. The 6809 ISA is technically defined as CISC but is much simpler than the larger and more complex x86-64 ISA. In fact, the 6809 ISA is simpler than many RISC ISAs, but because its arithmetic and logical instructions often use memory operands, it does not qualify for the RISC “load store” definition. It is easier to create pipelined micro-architectures for a load-store ISA, however the tradeoff is the requirement of an extra load or store instruction compared to the accumulator-based ISA of the 6809. This combined with coarsely encoded, fixed-width RISC instructions results in enlarged code size compared to the highly encoded variable length instructions of the 6809.

If we can decode these 6809 instructions as efficiently as RISC instructions, then we can leverage the code size advantage of the 6809’s instruction set. In this chapter we introduce μRTL, a custom microcode to RTL assembler, that can exploit the elegant structure and orthogonal design of the 6809 ISA. This creates an efficient translation layer that makes the Turbo9 implementation competitive and at times superior to modern RISC implementations. A traditional microcode assembler produces a large, sequentially ordered ROM to implement multicycle microarchitectures. μRTL takes a different approach by producing multiple synthesizable Verilog RTL modules with an emphasis on parallel rather than sequential decoding. Using the μRTL methodology, we created the Turbo9’s efficient decode stage that is part of the pipelined microarchitecture and is key to its performance.

Microprocessor Design Background

In order to explain the μRTL methodology, we will provide some background on microprocessor design. Whereas RISC instructions are designed for a pipelined implementation, the 6809 instruction set was designed for a multicycle implementation. We will therefore start with an example of a traditional multicycle implementation of the 6809 ISA that is more powerful but similar to Motorola's original implementation, then build on this example to show how we constructed the Turbo9's pipelined decode stage.

Example: Multicycle 6809 Microprocessor

A simplified high level block diagram of our theoretical multicycle 6809 microarchitecture is shown in Figure 3-1.

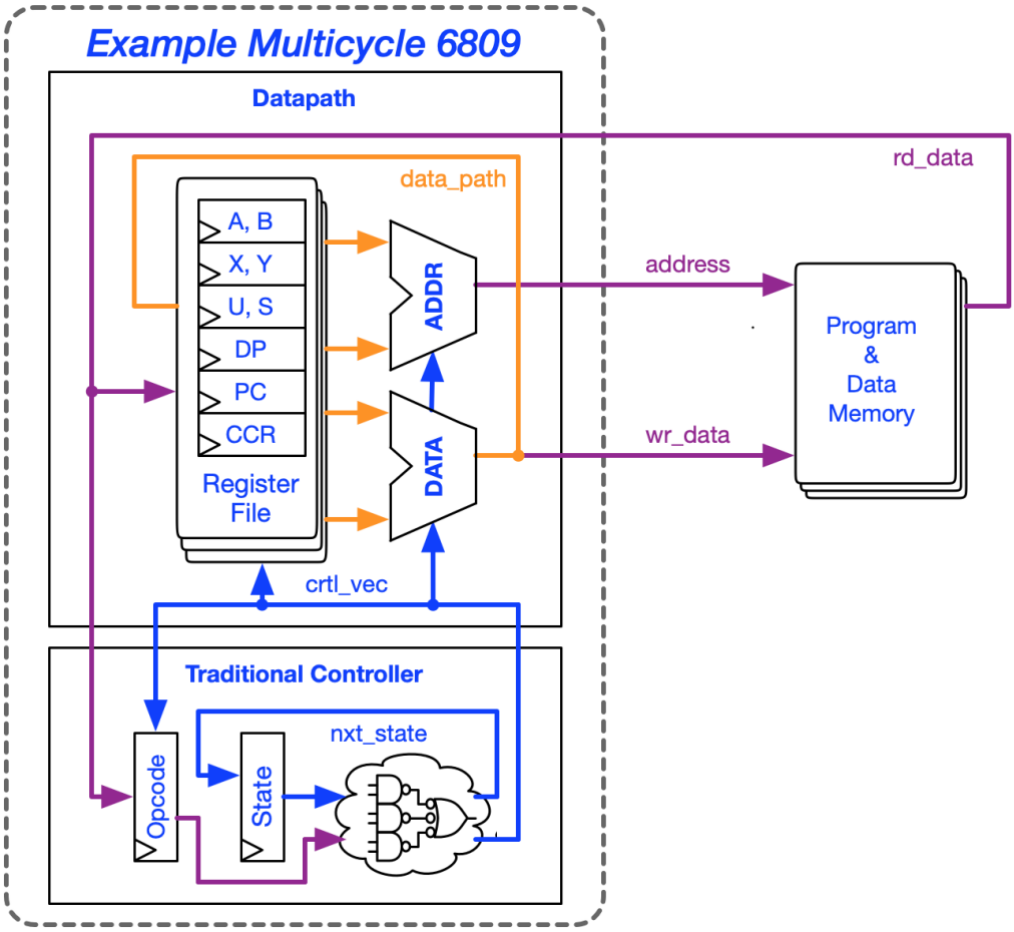


Figure 3-1. Block diagram of the example multicycle 6809 microarchitecture

The microprocessor is split into two main units: the data path where data is manipulated and the controller which controls the data path cycle by cycle. In this example the data path is made up of a register file and separate 16-bit data and address ALUs and a memory interface with the same capabilities as the Turbo9R. However, the controller is created with a traditional state machine. The state machine outputs control vectors *(crtl_vec)* to each element in the data path to set their function. In the previous chapter, some control vectors were presented for the various elements in the Turbo9 implementation.

The controller implements the sequences necessary to fetch, decode and execute instructions. Figure 3-2 shows all the states necessary to implement all different variations of the ADD instruction and addressing modes:

- Add Instructions:
  - ADDA: Add 8-bit memory with accumulator A
  - ADDB: Add 8-bit memory with accumulator B
  - ADDD: Add 16-bit memory with accumulator D
- Addressing Modes
  - Immediate Addressing
  - Direct Addressing
  - Index or Indirect Addressing
  - Extended Addressing

Each one of these addressing modes are implemented differently for 8-bit or 16-bit data. In the actual Motorola 6809 implementation, most of the addressing modes take much more than one cycle to access the data. For our example, we will assume each only takes one cycle given it has the same 16-bit memory bandwidth as the Turbo9R. Given three different accumulators and four different addressing modes, we require 12 different opcodes and map them to the opcode matrix also shown in Figure 3-2. Note how the other instructions are organized on the opcode matrix. There is an arranged order in the opcode encoding. For example, addressing modes are

encoded in the most significant nibble of the opcode which are represented by the columns in the opcode matrix. This structured opcode encoding will be important later in this chapter.

Page 1	Most Significant Nibble															
	DIR		REL		ACCA	ACCB	IDX / IND	EXT	IMM	DIR	IDX / IND	EXT	IMM	DIR	IDX / IND	EXT
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NEG	Page 2	BRA	LEAX	NEGA	NEGB	NEG	NEG	SUBA	SUBA	SUBA	SUBA	SUBB	SUBB	SUBB	SUBB
1		Page 3	BRN	LEAY					CMPA	CMPA	CMPA	CMPA	CMPB	CMPB	CMPB	CMPB
2		NOP	BHI	LEAS					SBCA	SBCA	SBCA	SBCA	SBCB	SBCB	SBCB	SBCB
3	COM	SYNC	BLS	LEAU	COMA	COMB	COM	COM	SUBD	SUBD	SUBD	SUBD	ADDD	ADDD	ADDD	ADDD
4	LSR	EMUL	BHS,BCC	PSHS	LSRA	LSRB	LSR	LSR	ANDA	ANDA	ANDA	ANDA	ANDB	ANDB	ANDB	ANDB
5		EMULS	BLO,BCS	PULS					BITA	BITA	BITA	BITA	BITB	BITB	BITB	BITB
6	ROR	LBRA	BNE	PSHU	RORA	RORB	ROR	ROR	LDA	LDA	LDA	LDA	LDB	LDB	LDB	LDB
7	ASR	LBSR	BEQ	PULU	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	CWAI	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	EXG	BGT				JMP	JMP	LDX	LDX	LDX	LDX	LDU	LDU	LDU	LDU
F	CLR	TFR	BLE	SWI	CLRA	CLRB	CLR	CLR		STX	STX	STX		STU	STU	STU

Least Significant Nibble

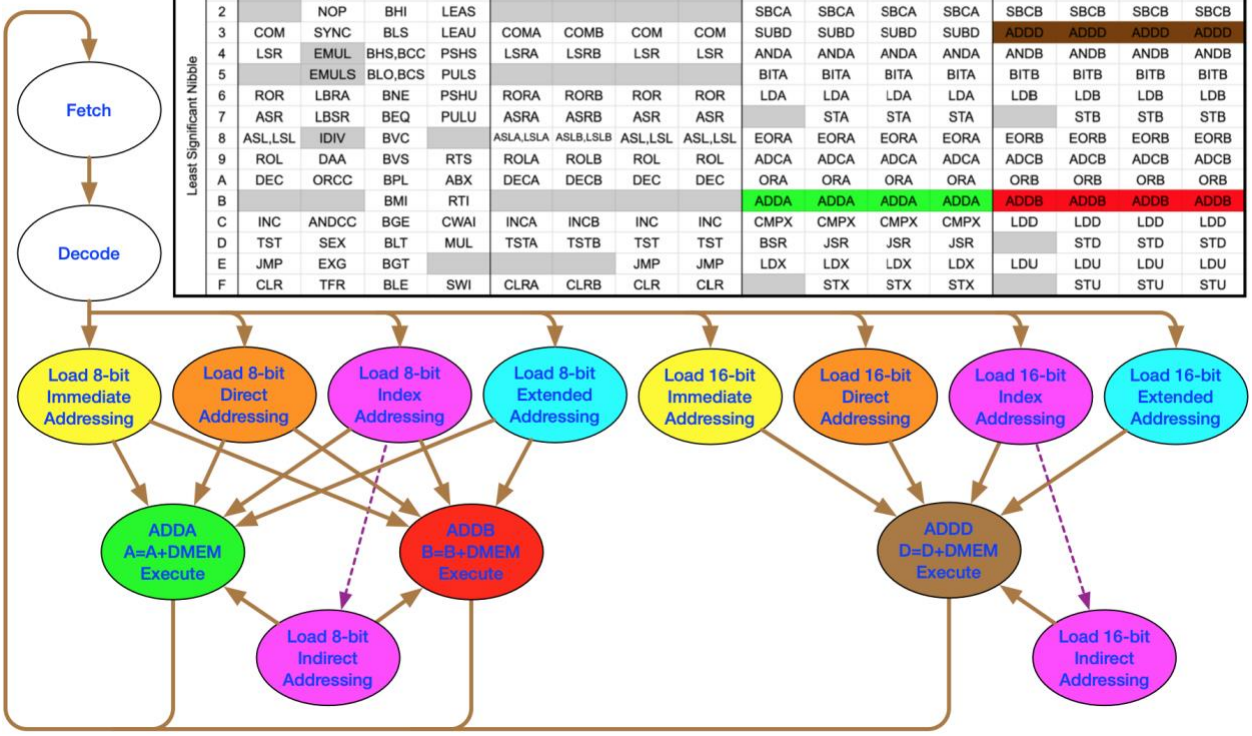


Figure 3-2. 6809 opcode matrix and state diagram of ADD instructions

Example: Microprogrammed State Machine

It should be apparent that designing the controller to decode the entire instruction set is a large task. While it is possible to accomplish this using a standard state machine coded in Verilog, the sheer size of the next-state and control vector logic makes the task difficult and the code maintenance even more challenging when written in RTL. So, it is common in CPU design to use a microprogrammed state machine to break down the controller design into a manageable hierarchy [7]. Understanding microprogramming concepts is important to μRTL methodology and to the design of the Turbo9's decode stage.

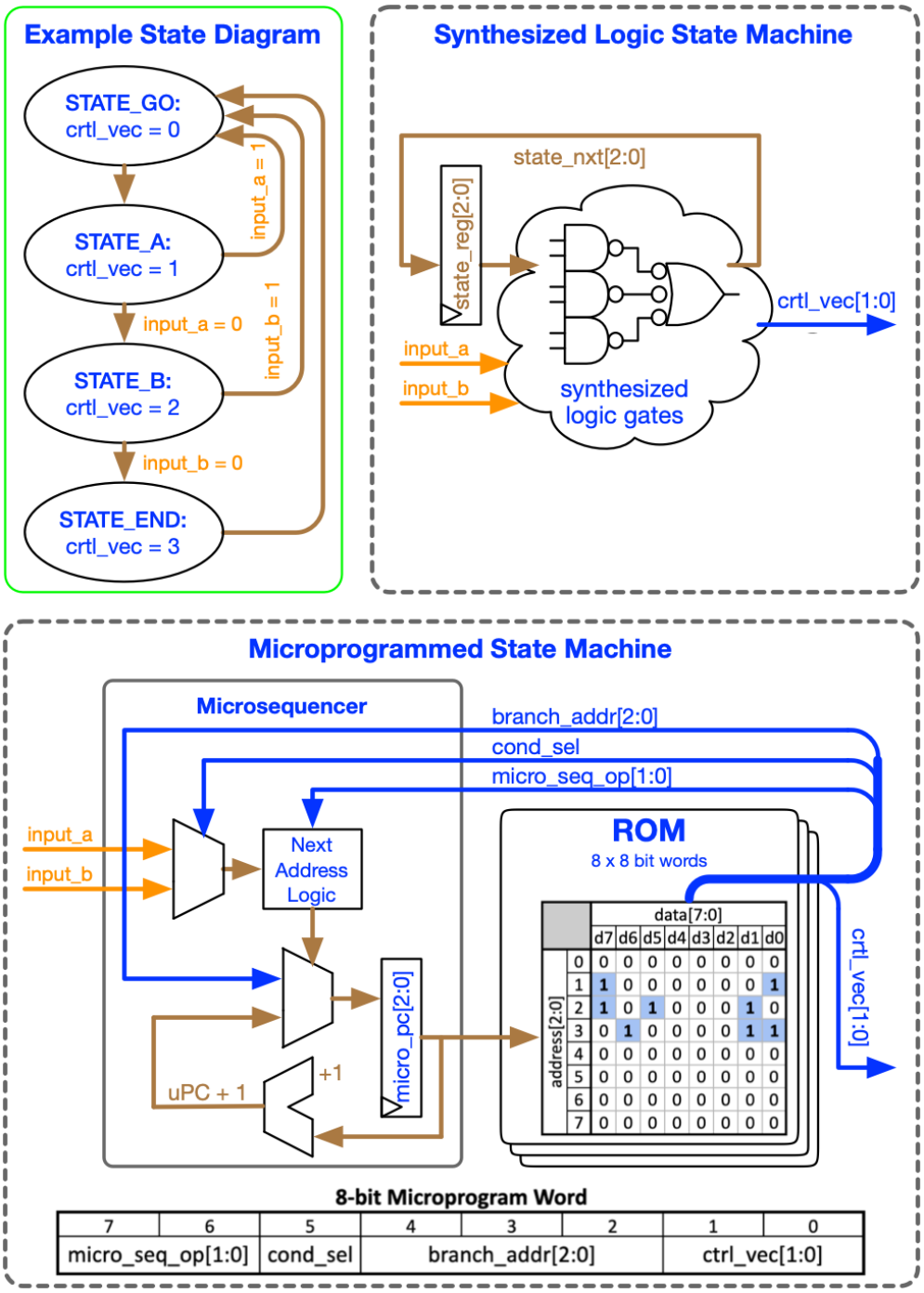


Figure 3-3. Synthesized state machine vs. microprogrammed state machine example

To give the reader a brief tutorial of microprograming, the implementation of a simple state diagram is presented in Figure 3-3. Though this example state diagram is trivial, the same techniques presented here can be used to implement the much larger fetch, decode, and execute

state diagram previously shown in Figure 3-2. Also shown in Figure 3-3 is the basic block diagram of a synthesized logic state machine. Note the 2-bit output control vector *(ctrl_vec[1:0]),* the inputs *(input_a, input_b),* the state register *(state_reg[2:0])* and finally synthesized logic with the next state *(state_nxt[2:0])* feeding back to the state register. The reader should be familiar with all these parts and understand how to design such a state machine to produce the example state diagram.

Now let us review the block diagram of the microprogrammed state machine. Notice the same inputs *(input_a, input_b)* and output *(ctrl_vec[1:0]),* but now the state register and logic have been replaced with a microsequencer and read only memory (ROM). The ROM's data output is now referred to as a "microprogram word." In this case the 8-bit microprogram word is further sliced into control vectors, one of which is the output *(ctrl_vec[1:0]).* More outputs can be easily added to the microprogrammed state machine by using a ROM with a wider microprogram word. In our multicycle 6809 microprocessor example, multiple control vectors would be used to set the function of the ALU, select registers for operands and result, as well as control all other necessary parts of the datapath in order to implement each cycle of a given instruction.

There are three other control vectors which are not used as outputs, but feedback to the microsequencer block. Before the advent of the single chip microprocessor, standard part microsequencer ICs were commonly used to build the central processing units of mainframe and mini-computers. In modern microprocessors, design engineers create their own custom microsequencers to suit the requirements of the microprocessor's controller. Our example microsequencer is simple, yet retains the functionality we need to convey. Common in all microprogrammed state machines is a register which stores the current ROM address. This is in

fact storing the "state" and serves the same purpose as the state register in the traditional logic state machine. In our example, we call this register the microprogram counter *(micro_pc[2:0]).* The *micro_pc* is fundamentally a register that can count sequentially or be loaded with a new value.

Table 3-1. Example microsequencer operation

Operation	micro_seq_op	cond_sel	input_a	input_b	next micro_pc
CONTINUE	00	x	x	x	micro_pc + 1
JUMP	01	x	x	x	branch_addr
JUMP IF (input_a)	10	0	0	x	micro_pc + 1
JUMP IF (input_a)	10	0	1	x	branch_addr
JUMP IF (input_b)	10	1	x	0	micro_pc + 1
JUMP IF (input_b)	10	1	x	1	branch_addr
undefined	11	x	x	x	undefined

This brings us back to the purpose of the three control vectors feeding back into the microsequencer: *micro_seq_op[1:0], cond_sel,* and *branch_addr[2:0].* These three control vectors determine the next value of the *micro_pc* and therefore the next microprogram word in the desired sequence. Refer to Table 3-1 for details regarding the operation of the example microsequencer.

Example: Microcode Assembler

Once we understand the basics of the microprogramed state machine hardware, we need to develop software tools to be able to efficiently create the sequences of microprogram words that become the contents of the microcode ROM. The most common way is to use a microcode assembler. As mentioned earlier, when mainframe and minicomputers were common, CPUs were built out of many discrete logic chips. Microsequencers could be purchased off the shelf from companies such as Texas Instruments, Advanced Micro Devices, and Altera. An assembler was provided to the customer to develop microcode for these parts [8]. With the dawn of the microprocessor age, the standardization of the discrete microsequencer became obsolete as

customized control units were built in-house by the microprocessor design teams. Since these microsequencers were custom-built for high performance, the microcode assembler became a custom affair as well. These tools are not publicly available, as a microprocessor's microsequencer and microprogram are usually considered to be proprietary information.

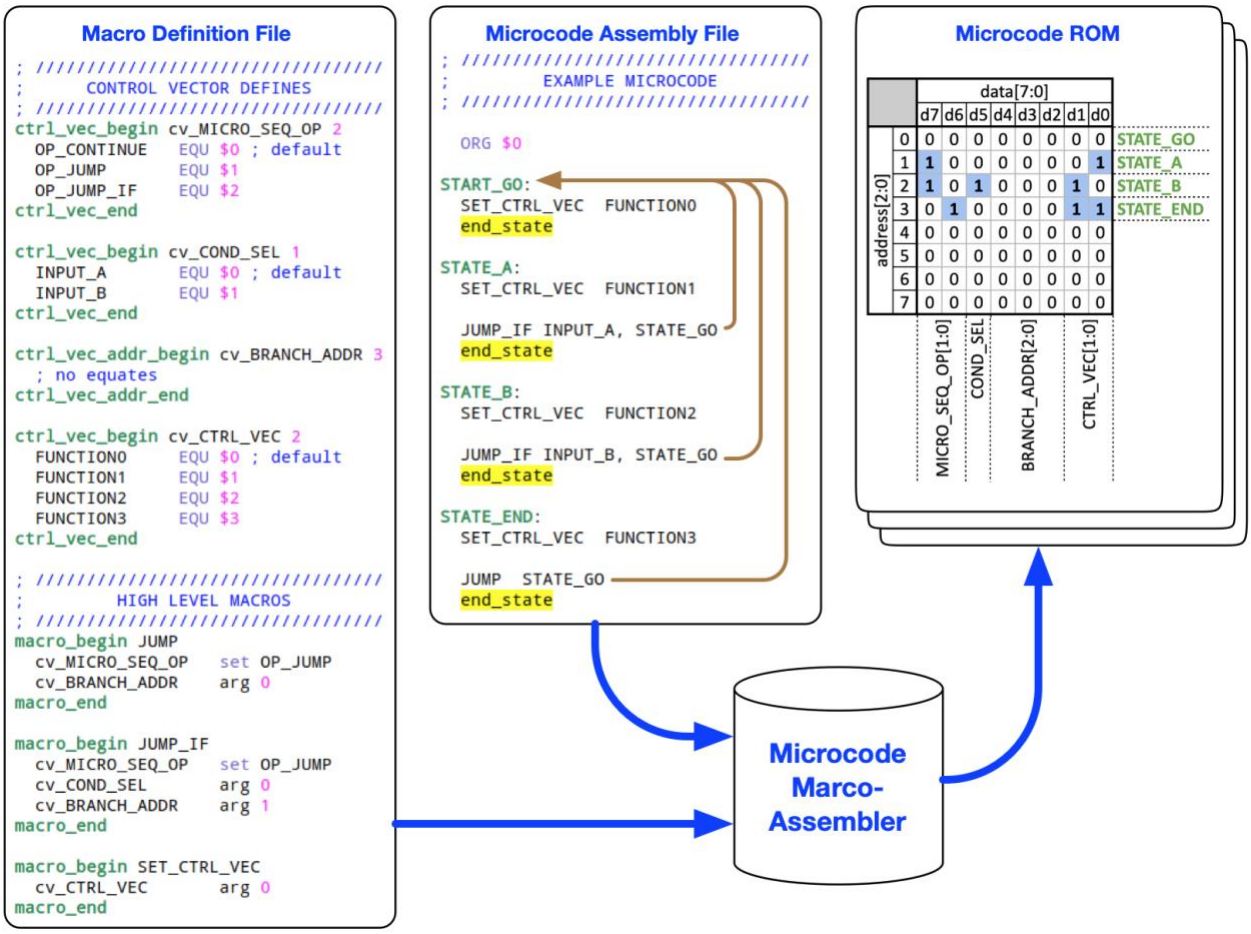


Figure 3-4. Example microcode & macro-based microcode assembler

The microcode we are presenting in Figure 3-4 actually uses the syntax we developed for our µRTL microcode assembler. We are using only the basic syntax (see Figure 3-5) for this example to maintain consistency throughout this document and develop the reader's familiarity with µRTL syntax. The new innovative features of µRTL will be presented in the next section.

It is common practice to use a macro-based assembler to develop microcode since a macro-assembler gives the design engineer the flexibility to adapt the assembler to an evolving micro-architecture during the design process. Figure 3-4 shows the toolchain flow for a macro based microcode assembler for our example design. The inputs to the macro-assembler are the microcode assembly file and the macro-definition file, which contains the definitions for each of the macros and control vector symbols used in the microcode assembly file. The microcode ROM output and how the control vectors are mapped to each microprogram word is also shown. The brown arrows on the microcode assembly file in Figure 3-4 match those in the state diagram in Figure 3-3 and are added to highlight the sequence of states and conditional branching for the reader.

Macro Definition Syntax

```
; //// Normal Control Vector Define /////
ctrl_vec_begin <cv_name> <bit_width>
  <symbol>  EQU <value> ; default value
  <symbol>  EQU <value>
  ...
ctrl_vec_end

; //// Address Control Vector Define ////
ctrl_vec_addr_begin <cv_name> <bit_width>
  ; labels in assembly file used
ctrl_vec_addr_end

; /////////// Marco Define //////////////
macro_begin <macro_name>
  <cv_name>   set   <symbol>
  <cv_name>   arg   <argument number>
  ...
macro_end
```

Microcode Assembly Syntax

```
; //////// Assembly Structure ///////////

  ORG <address>

LABEL_X:  ; LABEL_X = address
  <macro_name> <arg0> <arg1> <arg2> ...
  <macro_name> <arg0> <arg1> <arg2> ...
  ...
  end_state
  <macro_name> <arg0> <arg1> <arg2> ...
  <macro_name> <arg0> <arg1> <arg2> ...
  ...
  end_state
LABEL_Y:  ; LABEL_Y = address + 2
  <macro_name> <arg0> <arg1> <arg2> ...
  <macro_name> <arg0> <arg1> <arg2> ...
  ...
  end_state
```

Figure 3-5. Macro definition and microcode assembly basic syntax

The macro file contains the *ctrl_vec_begin* and *ctrl_vec_end* directives to define the control vectors and the symbolic name for each valid value. The *ctrl_vec_addr_begin* and *ctrl_vec_addr_end* directives are used to define a special control vector that can be assigned a microcode ROM address. All labels in the assembly file and their associated addresses are

automatically defined for this control vector. The macro definition describes which control vectors are set by a constant or a macro argument.

The microcode assembly is performed by using the macros to set the control vectors in each state. The *end_state* directive acts as the delimiter between states. Those familiar with microprocessor assembly usually interpret one instruction as one mnemonic, whereas in this microcode assembly example multiple lines of macros make up one microprogram word. Whenever the assembler reaches an *end_state* directive, it assembles all the control vector values into the current microprogram word, increments the address, and then resets all the control vectors back to their default value that are then modified by the macros in the following state. The *ORG <address>* directive sets the address of the current microprogram word. Labels are inferred as symbols set to an address that can be assigned to address control vectors.

µRTL Based Pipelined Architecture

In the Turbo9, a pipelined micro architecture is used to overlap the fetch, decode and execute states of adjacent instructions. The fetch, decode and execute stages can be viewed as separate digital machines that are each working on a different instruction at a different state of the multi-cycle state diagram. This instruction level parallelism increases the throughput of the microprocessor, with some instructions able to achieve single cycle execution from the viewpoint of the programmer. Figure 3-6 shows how the instructions A, B, and C are mapped to Turbo9's microarchitecture while it is processing the fetch, decode and execute states of three multicycle state diagrams in parallel.

The Turbo9's 6809 based ISA uses tightly encoded variable-length instructions that do not map directly to the control vectors connected to the data path in the execute stage. Additionally, some instructions require sequences of multiple micro-ops. The decode stage therefore needs to implement a CISC to RISC translation layer. The reader may have realized

that the micro-operations in the microprogrammed state machine earlier are actually a wide RISC instruction. RISC instructions and micro-ops are both fixed-length instructions that are loosely encoded and contain bit-fields that control a datapath directly. The micro-op tends to be wider with more parallel control vectors, but could be defined as a RISC instruction.

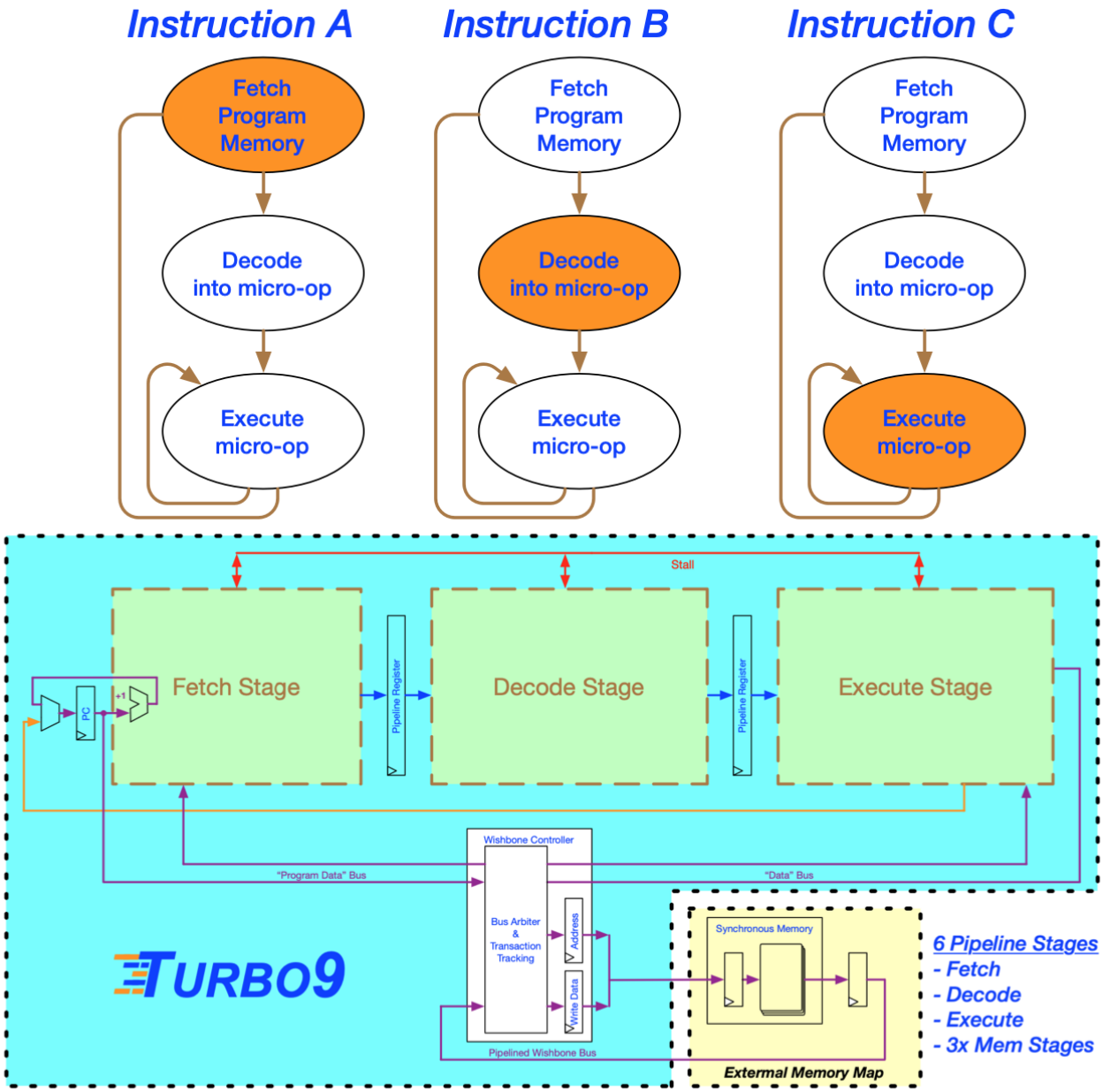


Figure 3-6. Illustration of instruction parallelism in the Turbo9's pipeline

It would therefore be a wise choice to include some form of microprogrammed state machine in the decode stage to generate these RISC-like micro-ops, given the similarities. However, while one could use a brute-force method of writing all the microcode sequences and assembling them into a ROM, it would not be the most efficient in terms of area and cycle-by-cycle performance. Referring back to Figure 3-2, it was shown that the 6809's instruction set is very orthogonal and much of the information necessary to execute the instruction is encoded directly in the opcode. This information includes the addressing mode, data operands, and address operands. It is to our advantage to directly decode from the opcode as much as possible. This will then reduce the width of the microprogram word by removing control vectors and also reduce the overall number of microprogram states necessary to implement the 6809 ISA. Our new μRTL assembler provides the methodology to extract these addressing modes, data operands and address operands from the tightly encoded 6809-based instruction set shown in Figure A-1 of Appendix A. The resulting decode stage uses the minimal number of sequential states, maximum amount of direct decoding, and uses synthesized logic rather than a ROM to efficiently produce these RISC-like micro-ops.

We now have enough background to present the details of the Turbo9's decode stage in Figure 3-7. The reader should already recognize some familiar architecture. For now, let us set aside the microsequencer, microcode μRTL, and μRTL decode blocks and cover the supporting blocks first.

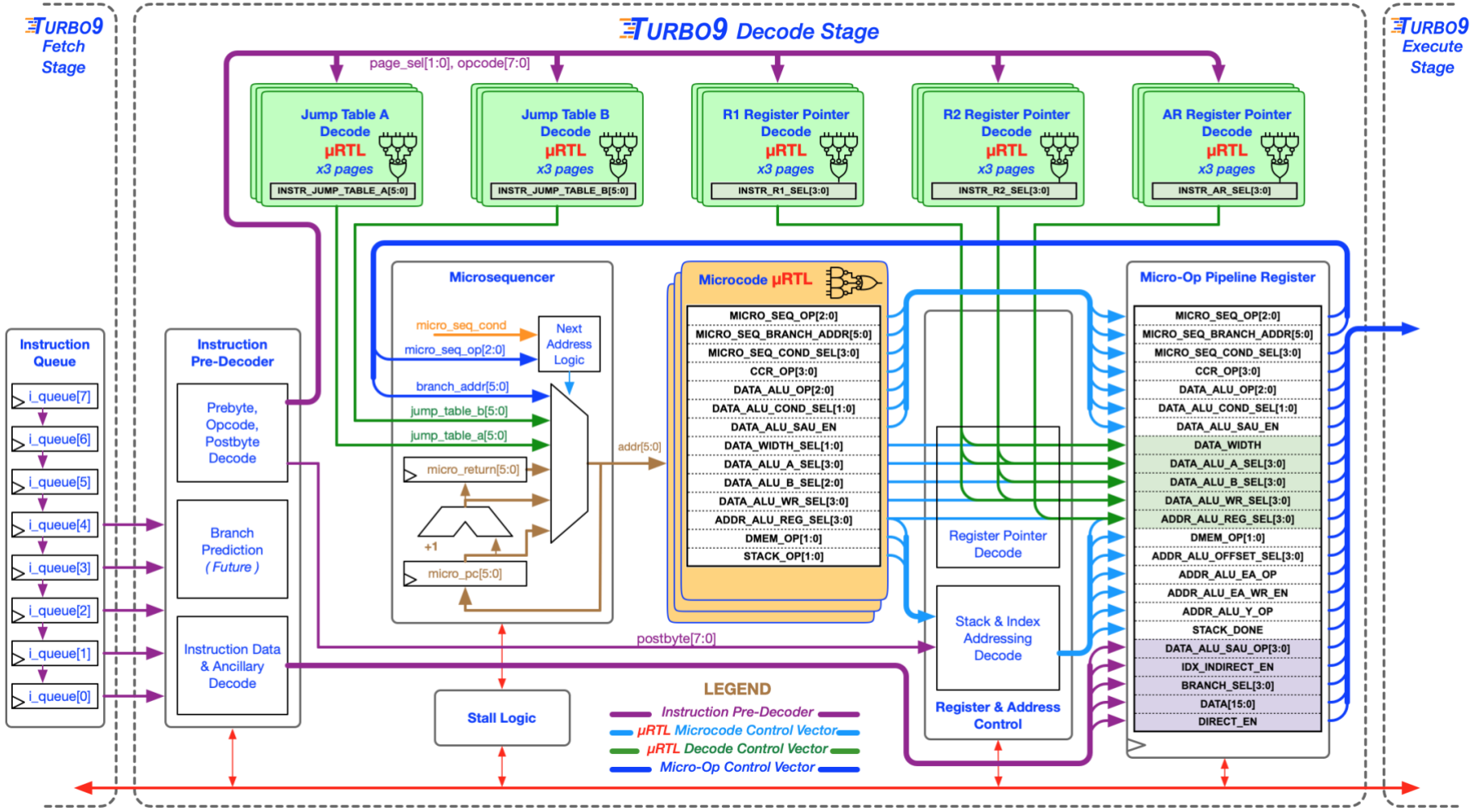


Figure 3-7. Detailed block diagram of the Turbo9 decode stage with µRTL generated blocks

Micro-Op Pipeline Register

This is the pipeline register which contains the micro-op delivered to the execute stage. Table 3-2 lists the description and the decode logic source for each control vector in the Micro-Op Pipeline Register.

Table 3-2. Turbo9 micro-operation control vectors & decode logic source

Micro-Op Control Vector	Instr. Pre-Decode	Stack & Indexed Postbyte Decode	Micro-code Control Vector	Reg Pointer µRTL Decode	Description
MICRO_SEQ_OP[2:0]			X		Microsequencer operation
MICRO_SEQ_BRANCH_ADDR[5:0]			X		Microsequencer branch address
MICRO_SEQ_COND_SEL[3:0]			X		Microsequencer condition select
CCR_OP[3:0]			X		Condition code register operation
DATA_ALU_OP[2:0]			X		Data ALU operation
DATA_ALU_COND_SEL[1:0]			X		Data ALU condition input select
DATA_ALU_SAU_EN			X		Sequential arithmetic unit enable
DATA_WIDTH		X	X	R1	Data width for ALU flags & memory
DATA_ALU_A_SEL[3:0]		X	X	R1 | R2	Data ALU "A operand" select
DATA_ALU_B_SEL[3:0]			X	R2	Data ALU "B operand" select
DATA_ALU_WR_SEL[3:0]		X	X	R1 | R2	Data ALU "write operand" select
ADDR_ALU_REG_SEL[3:0]		X	X	AR	Address ALU register select
DMEM_OP[1:0]		X	X		Data memory operation
ADDR_ALU_OFFSET_SEL[3:0]		X			Address ALU offset select
ADDR_ALU_EA_OP		X			Effective address operation
ADDR_ALU_EA_WR_EN		X			Effective address write enable
ADDR_ALU_Y_OP		X			Address ALU operation
STACK_DONE		X			Stack done condition flag
DATA_ALU_SAU_OP[3:0]	X				Sequential arithmetic unit operation
IDX_INDIRECT_EN	X				Index addressing indirect enable
BRANCH_SEL[3:0]	X				Branch instruction condition select
IDATA[15:0]	X				Instr. data (immediate data or address)
DIRECT_EN	X				Direct address enable

Referring back to Figure 3-7, the control vectors highlighted in green are a product of decoding directly from the opcode via the µRTL decode tables, but can also be modified cycle-by-cycle by the microcode µRTL block. The control vectors highlighted in purple are also decoded directly off the opcode, but remain constant across all micro-ops in one instruction.

Finally, the remaining unhighlighted control vectors are dependent only on the microcode μRTL block.

Register and Address Control Block

The final construction of the micro-op before it is placed in the pipeline register occurs in the register and address control block. The register pointer decode logic determines whether the data operands and address operands are derived from microcode, or a register pointer is used from the μRTL decode tables. The other major function of the register and address control block is to decode the indexed addressing modes and stack operations encoded in the instruction's postbyte. The stack instructions use a simple bitmask for each of the 6809's registers. The 6809's indexed addressing modes are encoded in a very organized manner illustrated in the postbyte matrix shown in Figure 3-8. This decode logic is relatively straight forward to hand-code in Verilog RTL and therefore does not require a μRTL decode block.

Indexed Address Postbyte			Most Significant Nibble															
			X	X	Y	Y	U	U	S	S	X	X	Y	Y	U	U	S	S
			0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
			0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Least Significant Nibble	0000	0	0,X	-16,X	0,Y	-16,Y	0,U	-16,U	0,S	-16,S	,X+	[,X+]	,Y+	[,Y+]	,U+	[,U+]	,S+	[,S+]
	0001	1	1,X	-15,X	1,Y	-15,Y	1,U	-15,U	1,S	-15,S	,X++	[,X++]	,Y++	[,Y++]	,U++	[,U++]	,S++	[,S++]
	0010	2	2,X	-14,X	2,Y	-14,Y	2,U	-14,U	2,S	-14,S	,-X	[,-X]	,-Y	[,-Y]	,-U	[,-U]	,-S	[,-S]
	0011	3	3,X	-13,X	3,Y	-13,Y	3,U	-13,U	3,S	-13,S	,--X	[,--X]	,--Y	[,--Y]	,--U	[,--U]	,--S	[,--S]
	0100	4	4,X	-12,X	4,Y	-12,Y	4,U	-12,U	4,S	-12,S	,X	[,X]	,Y	[,Y]	,U	[,U]	,S	[,S]
	0101	5	5,X	-11,X	5,Y	-11,Y	5,U	-11,U	5,S	-11,S	B,X	[B,X]	B,Y	[B,Y]	B,U	[B,U]	B,S	[B,S]
	0110	6	6,X	-10,X	6,Y	-10,Y	6,U	-10,U	6,S	-10,S	A,X	[A,X]	A,Y	[A,Y]	A,U	[A,U]	A,S	[A,S]
	0111	7	7,X	-9,X	7,Y	-9,Y	7,U	-9,U	7,S	-9,S	D,X	[D,X]	D,Y	[D,Y]	D,U	[D,U]	D,S	[D,S]
	1000	8	8,X	-8,X	8,Y	-8,Y	8,U	-8,U	8,S	-8,S	8bit,X	[8bit,X]	8bit,Y	[8bit,Y]	8bit,U	[8bit,U]	8bit,S	[8bit,S]
	1001	9	9,X	-7,X	9,Y	-7,Y	9,U	-7,U	9,S	-7,S	16bit,X	[16bit,X]	16bit,Y	[16bit,Y]	16bit,U	[16bit,U]	16bit,S	[16bit,S]
	1010	A	10,X	-6,X	10,Y	-6,Y	10,U	-6,U	10,S	-6,S	D,X	[D,X]	D,Y	[D,Y]	D,U	[D,U]	D,S	[D,S]
	1011	B	11,X	-5,X	11,Y	-5,Y	11,U	-5,U	11,S	-5,S	D,X	[D,X]	D,Y	[D,Y]	D,U	[D,U]	D,S	[D,S]
	1100	C	12,X	-4,X	12,Y	-4,Y	12,U	-4,U	12,S	-4,S	8bit,PC	[8bit,PC]	8bit,PC	[8bit,PC]	8bit,PC	[8bit,PC]	8bit,PC	[8bit,PC]
	1101	D	13,X	-3,X	13,Y	-3,Y	13,U	-3,U	13,S	-3,S	16bit,PC	[16bit,PC]	16bit,PC	[16bit,PC]	16bit,PC	[16bit,PC]	16bit,PC	[16bit,PC]
	1110	E	14,X	-2,X	14,Y	-2,Y	14,U	-2,U	14,S	-2,S	8bit	[8bit]	8bit	[8bit]	8bit	[8bit]	8bit	[8bit]
	1111	F	15,X	-1,X	15,Y	-1,Y	15,U	-1,U	15,S	-1,S	16bit	[16bit]	16bit	[16bit]	16bit	[16bit]	16bit	[16bit]

Figure 3-8. Postbyte matrix for indexing addressing modes

Instruction Pre-Decoder

The instruction pre-decoder is the interface to the fetch stage. The prebyte, opcode and postbyte decode logic determines the instruction format and outputs the necessary signals to various blocks in the decode stage. Its top priority is to determine whether the number of valid

bytes in the instruction queue is equal to or greater than the length of the instruction. If this condition is not true and the micro sequencer is requesting the next pre decoded instruction, then the stall logic will stall the necessary stages in the pipeline until the full instruction is available from the queue. The prebyte is decoded into the page select *(page_sel[1:0])* and sent to the μRTL decode blocks. The postbyte signal *(postbyte[7:0])* is sent to the stack and index addressing decode logic. The instruction data and ancillary decode logic derives some basic control vectors directly from the opcode, the most important of which is the instruction data control vector *(IDATA[15:0])* which contains any data that is embedded into the instruction. In the case of instructions that use immediate addressing, this control vector would contain the immediate data to be used as an operand in the execute stage. For instructions that use indexed addressing, the control vector will contain the offset and will be used as an address operand in the execute stage. Finally, the branch prediction logic shown here has not been implemented yet, but is planned for the future.

Microsequencer

The custom microsequencer that we designed for the Turbo9's decode stage is relatively simple and adds just a few features to the microsequencer that we presented in the earlier example. There are more sources of branch addresses including a subroutine return address register *(micro_return[5:0])* and jump vectors from the μRTL jump tables. Also note the *micro_pc* register is now placed as an input to the next address mux rather than at the output. This allows signals from the instruction pre-decoder to flow through the μRTL jump tables, through the microcode block and into the micro-op pipeline register within one clock cycle. We have monitored this propagation path to ensure it is similar to the worst-case propagation path in the execute stage. Further pipelining could be used to break up the decode stage and execute stage, but this is unnecessary since our propagation paths are well balanced between all the

stages and we are more than happy with our maximum frequency results. Last, the *micro_seq_cond* signal comes from the Condition Code Register (CCR) logic in the execute stage. Figure 3-9 shows the Turbo9 microsequencer operation control vector definition.

```
; ////////////////////////////////////////// cv_MICRO_SEQ_OP definition
ctrl_vec_begin cv_MICRO_SEQ_OP 3
  OP_CONTINUE               EQU $0 ; u_pc = u_pc+1
  OP_JUMP                   EQU $1 ; u_pc = BRANCH_ADDR
  OP_CALL                   EQU $2 ; u_pc = BRANCH_ADDR, u_return = u_pc+1
  OP_RETURN                 EQU $3 ; u_pc = u_return
  OP_JUMP_TABLE_B           EQU $4 ; u_pc = JUMP_TABLE_B
  OP_JUMP_TABLE_A_NEXT_PC   EQU $5 ; u_pc = JUMP_TABLE_A
ctrl_vec_end
; //////////////////////////////////////////
```

Figure 3-9. Turbo9 microsequencer operation control vector µRTL microcode definition

µRTL Assembler

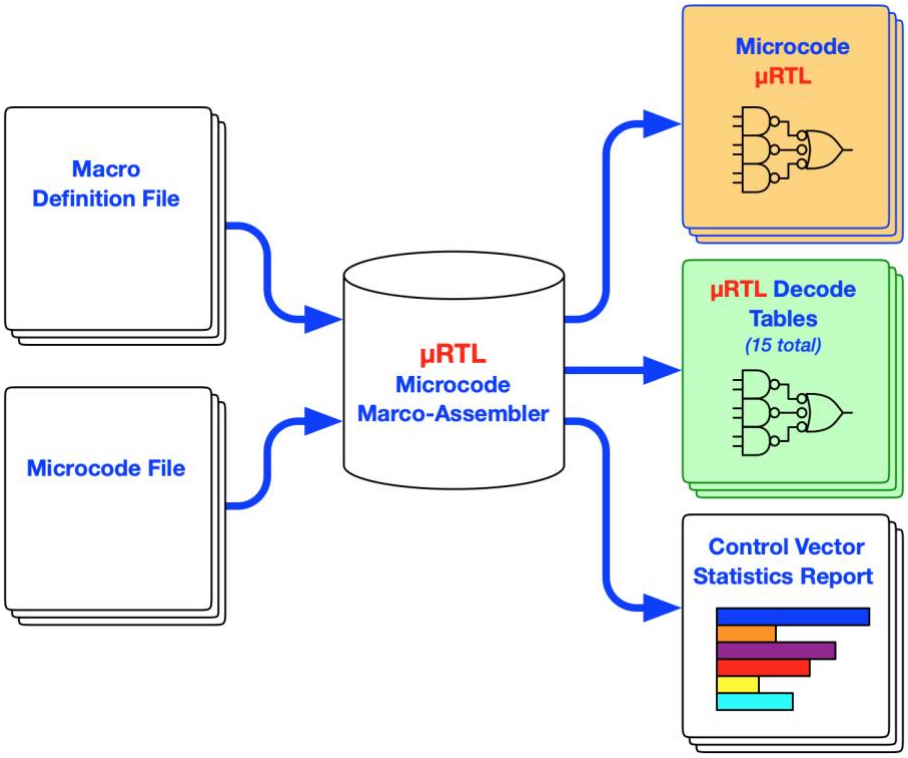


Figure 3-10. µRTL microcode macro-assembler tool flow

Up to this point, we have introduced all of the blocks in the decode stage that are hand coded in Verilog RTL. The remaining µRTL generated blocks will perform the bulk of the translation from each of the Turbo9's instructions into the control vectors within each micro-op.

Figure 3-10 shows the same input files as the basic microcode assembler example, but now the outputs are different. There is no longer a microcode ROM, instead the microcode is assembled into a Verilog RTL file. A ROM is no longer used because the microprogram size is greatly reduced due to the efficient partitioning and separation of optimizable logic into the fifteen decode tables. The µRTL decode tables are Verilog modules that are generated by the assembler. These Verilog blocks are easily synthesized into minimal area blocks by modern synthesis tools. Finally, µRTL creates a report of statistical data for every single control vector, which gives the engineer feedback to optimize their design.

µRTL Generated Verilog Blocks

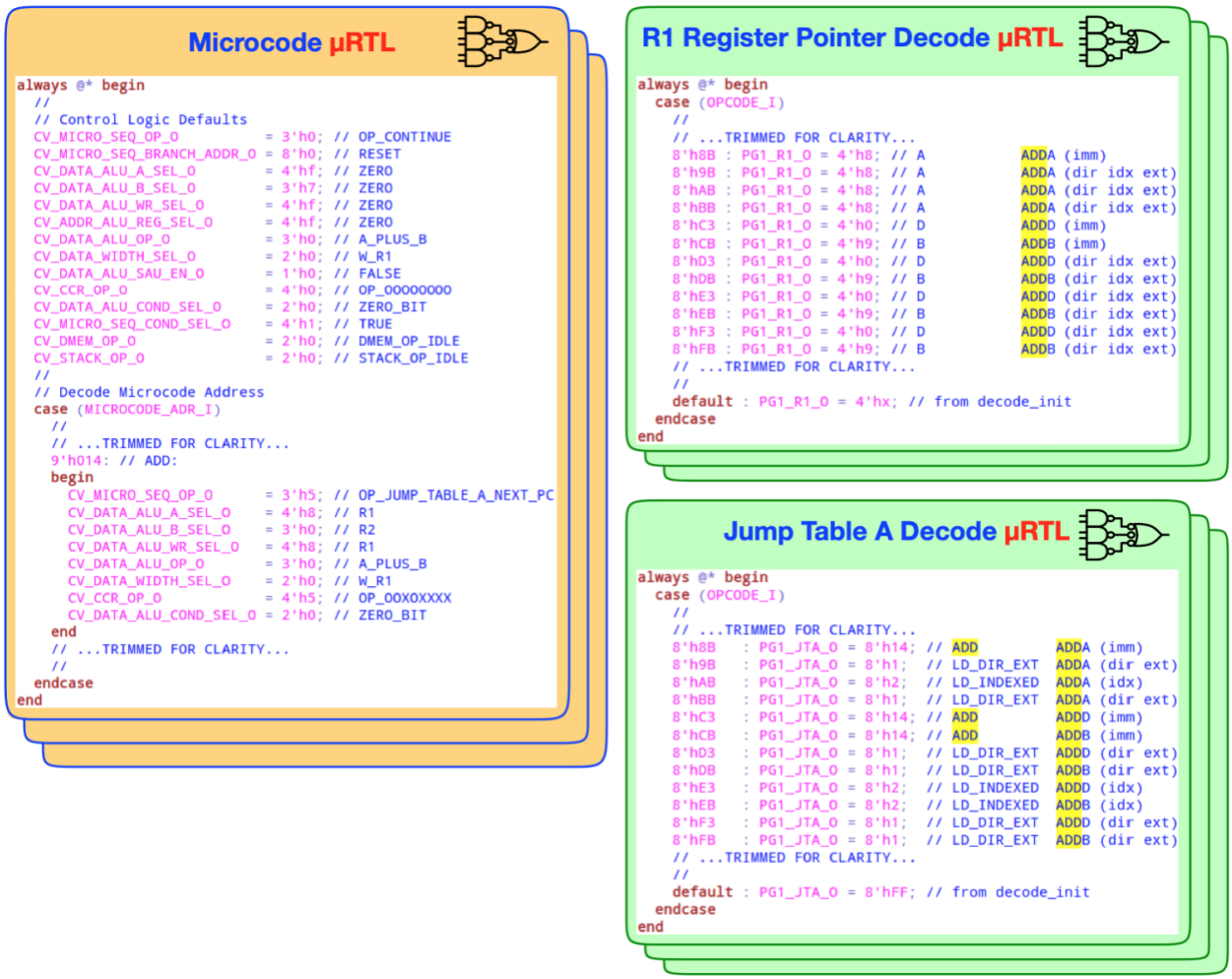


Figure 3-11. µRTL generated Verilog outputs

Code segments of the microcode µRTL and decode µRTL are shown in Figure 3-11. The decode tables are simply created using case statements for a given opcode. The entries in the case statements all include comments from the microcode. In this case, the default is set to "don't care" (4'hx) by the *decode_init* directive in the microcode. This allows the synthesis tool to optimize all unused entries in the opcode matrix. The microcode µRTL is similar to the decode blocks, but its address input is provided by the microsequencer, and it outputs multiple control vectors. Note that these control vector assignments in each case statement block also have comments taken from the microcode. Not shown is the listing of the microcode in comments with each microprogram word in the Verilog RTL. All this together allows someone not familiar with microcode to read the Verilog output as they would a standard list file from an assembler.

µRTL Statistics Report

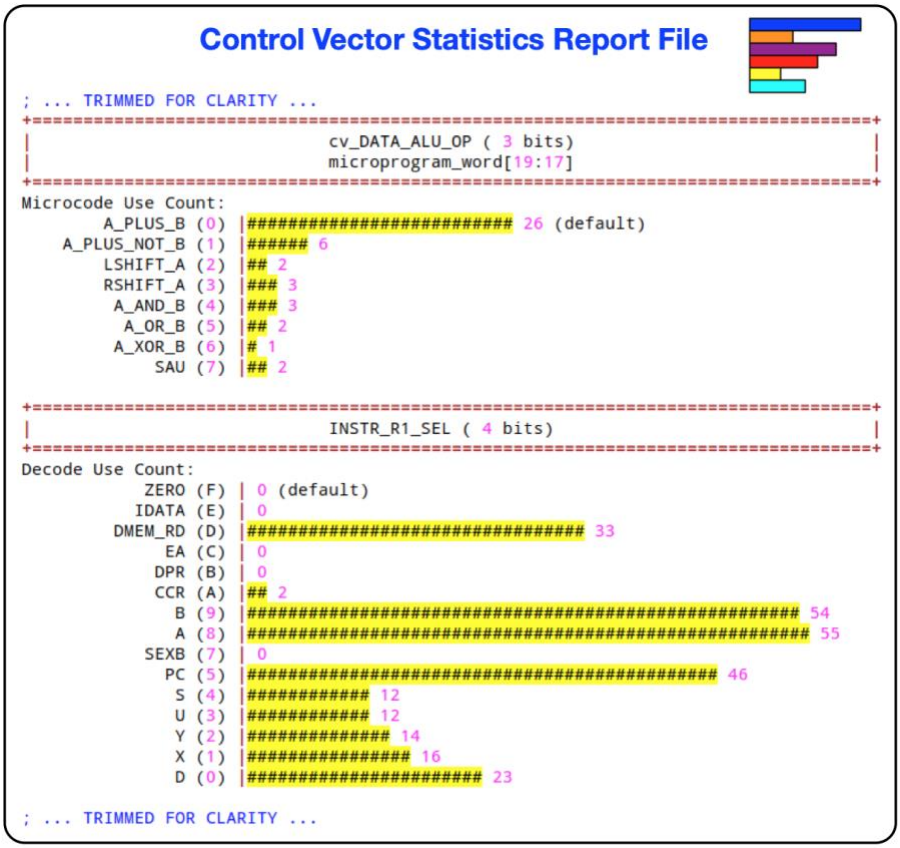
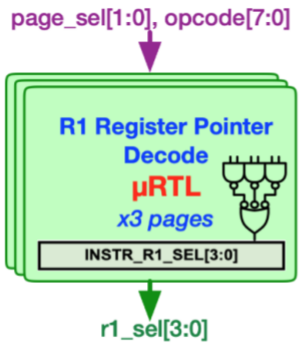


Figure 3-12. µRTL generated control vector statistics report file

The control vector statistics provided by µRTL is a advanced new feature and offers valuable insight into the design. It is useful for optimizing both the decode stage and the execute stage since the control vectors are the interface between these two stages. A report file is created with formatted usage data of each symbol associated with each control vector. It creates this data for both the decode tables and the microcode. Control vectors can be set in either the microcode or the decode tables, or both. Using this data, the designer can recognize when a symbol is not being used and could change the encoding of that control vector for better optimization. These types of optimizations directly affect the area of the resulting design as well as the static timing analysis. Since we are targeting synthesized logic gates rather than a ROM, having this data available to the design engineer is critical.



267 opcodes mapped to 10 registers

Target Reg	#	Description
A	55	A Accumulator
B	54	B Accumulator
PC	46	Program Counter
DMEM_RD	33	Data Memory Read
D	23	D Accumulator
X	16	X Index Register
Y	14	Y Index Register
U	12	U Stack Pointer
S	12	S Stack Pointer
CCR	2	Condition Code Reg

Page 1	Most Significant Nibble															
Least Significant Nibble	DIR 0	1	REL 2	3	ACCA 4	ACCB 5	IDX / IND 6	EXT 7	IMM 8	DIR 9	IDX / IND A	EXT B	IMM C	DIR D	IDX / IND E	EXT 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

Figure 3-13. R1 register pointer µRTL decode opcode matrix

An example of how the control vector statistics is useful is shown in Figure 3-13. This figure shows a visualized version of the statistics for the R1 register pointer decode block. Appendix B shows the complete statistics for all the jump tables and all the register pointer tables across the entire opcode matrix. It should be apparent to the reader by looking at the colorization that this generated RTL block can be greatly minimized using logic synthesis tools. If this register pointer data was part of the microcode and not separated into its own decode block, this advantage of direct decode of the opcode would be lost, and therefore the area of the design would grow.

Example: Pipelined Turbo9 Microcode

Macro Definition File

```
; control vector defines not shown

macro_begin DATA_ADD
  cv_DATA_ALU_COND_SEL  set ZERO_BIT
  cv_DATA_ALU_OP        set A_PLUS_B
  cv_DATA_ALU_A_SEL     arg 0
  cv_DATA_ALU_B_SEL     arg 1¤
macro_end

macro_begin DATA_WRITE
  cv_DATA_ALU_WR_SEL  arg 0
macro_end

macro_begin SET_DATA_WIDTH
  cv_DATA_WIDTH_SEL   arg 0
macro_end

macro_begin CCR_OP_W
  cv_CCR_OP     arg 0
macro_end

macro_begin JUMP_TABLE_A_NEXT_PC
  cv_MICRO_SEQ_OP     set OP_JUMP_TABLE_A_NEXT_PC
macro_end
```

New µRTL Decode Directive!

```
; Initialize Decode Table:
decode_init <tablename> <ctrl_vec> <default> ; Comment
decode_init pg1_JTA cv_MICRO_SEQ_BRANCH_ADDR FF ;Example

; Add Decode Table Entry:
decode <tablename> <equ> <opcode0...opcodeN> ; Comment
decode pg1_JTA ABX $3A ; ABX(inh) ; Jump Vector Example
```

Microcode File

```
; ////////////////////////////////////////// ADD
; //
ADD:
  decode pg1_JTA ADD     $8B          ; ADDA (imm)
  decode pg1_R1  A       $8B          ; ADDA (imm)
  decode pg1_R2  IDATA   $8B          ; ADDA (imm)

  decode pg1_JTA ADD     $CB          ; ADDB (imm)
  decode pg1_R1  B       $CB          ; ADDB (imm)
  decode pg1_R2  IDATA   $CB          ; ADDB (imm)

  decode pg1_JTA ADD     $C3          ; ADDD (imm)
  decode pg1_R1  D       $C3          ; ADDD (imm)
  decode pg1_R2  IDATA   $C3          ; ADDD (imm)

  decode pg1_JTB ADD     $9B $AB $BB ; ADDA (dir idx ext)
  decode pg1_R1  A       $9B $AB $BB ; ADDA (dir idx ext)
  decode pg1_R2  DMEM_RD $9B $AB $BB ; ADDA (dir idx ext)

  decode pg1_JTB ADD     $DB $EB $FB ; ADDB (dir idx ext)
  decode pg1_R1  B       $DB $EB $FB ; ADDB (dir idx ext)
  decode pg1_R2  DMEM_RD $DB $EB $FB ; ADDB (dir idx ext)

  decode pg1_JTB ADD     $D3 $E3 $F3 ; ADDD (dir idx ext)
  decode pg1_R1  D       $D3 $E3 $F3 ; ADDD (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 ; H is masked for 16bit

  JUMP_TABLE_A_NEXT_PC
  end_state
```

Figure 3-14. µRTL microcode macro definition and microcode file

We can now take a closer look at a sample of the microcode and macro definition files used for implementation of the ADD instructions presented earlier in Figure 3-2. The execute

state of the add instruction is shown in Figure 3-14, along with the macros used (for clarity, control vector defines are not shown). This microcode executes all the variations of the add instruction (ADD) shown previously.

Note the new decode directive. This directive is what makes μRTL so powerful by allowing it to balance parallel vs. sequential decoding of instructions. Any control vector that is only dependent on the opcode can be decoded efficiently and directly from the opcode. By identifying the control vectors that do not need be decoded in the microcode, we can consolidate several microprogram words into one. In the sample code shown, register pointer R2 is added to register pointer R1 and stored in register pointer R1. The decode tables then define what registers R1 and R2 point to, depending on the opcode. For example, the ADDA Immediate instruction sets the R1pointer to the A register and the R2 pointer is set to the immediate data passed through the *IDATA* field in the micro-op. For the direct indexed or extended addressing modes of this same instruction, R2 points to the DMEM_RD register which is loaded in a previous state that implements the target memory access.

Register pointers are just one way to use decode tables. The jump tables *(pg1_JTA* and *pg1_JTB)* are used to string together different sequences of micro-ops depending on the opcode. This allows us to implement the different addressing modes needed for the instructions. The immediate addressing versions of the ADD instruction use jump table A (pg1_JTA) to reach this micro-op directly because the immediate is already available in the *IDATA* field of the micro-op. This allows us to accomplish the immediate ADD instruction in one clock cycle. Note, the other addressing modes reach this microprogramed state via jump table B, having already used jump table A for the addressing mode.

If this was a multicycle processor, we would jump to the fetch state after completing this execute state. However, since this is a pipelined design, we can assume that the next instruction has already been fetched and decoded. Therefore, the *JUMP_TABLE_A_NEXT_PC* directs the micro sequencer to vector to the jump table A address pointed to by the next instruction.

Pipelined Turbo9 Cycle-By-Cycle Performance

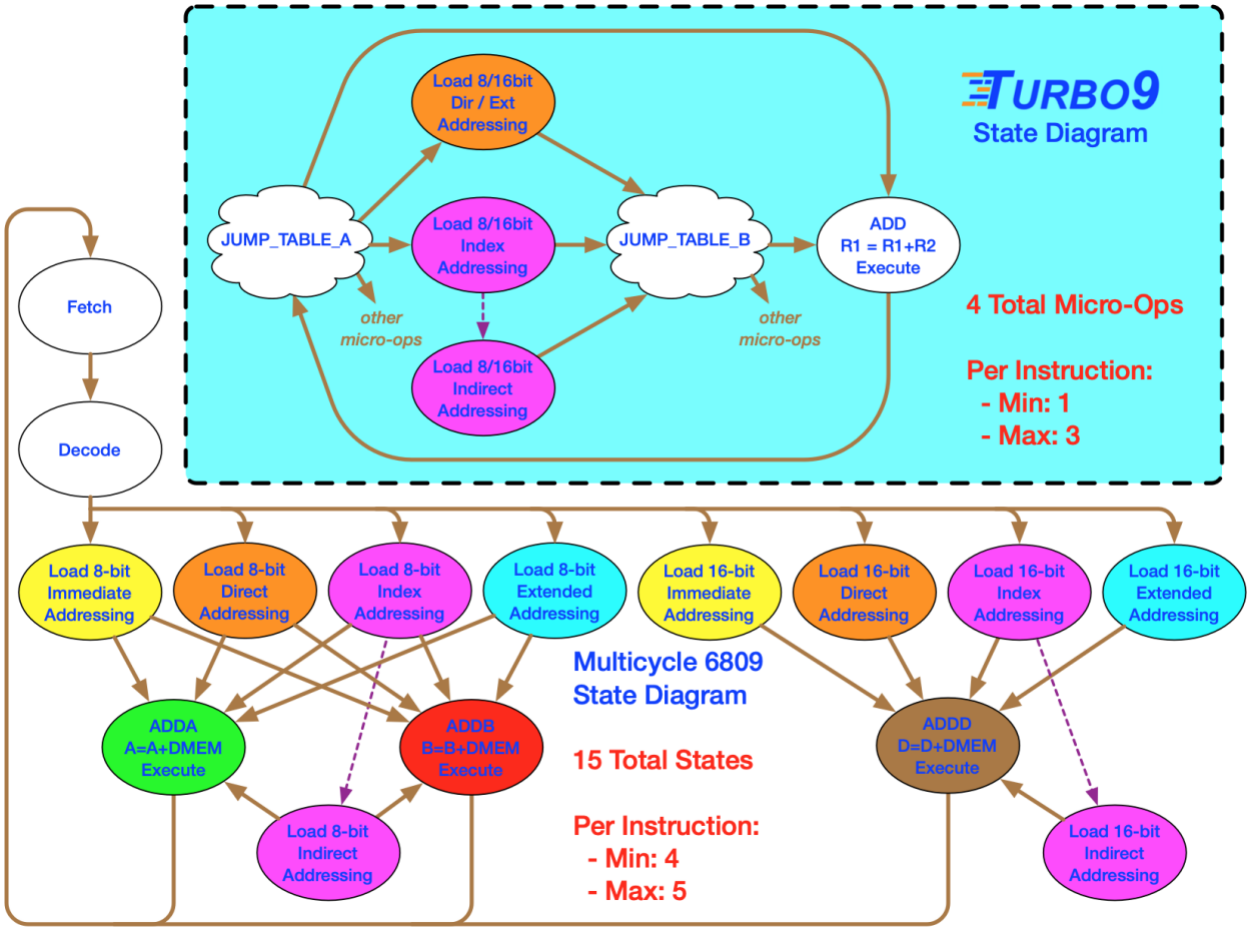


Figure 3-15. State diagram comparison: Turbo9 vs. example multicycle 6809

We're now in a position to compare the execution of the Turbo9 vs our traditional multicycle 6809 presented at the beginning of this chapter. Presented in Figure 3-15 is the Turbo9's state diagram versus the multicycle state diagram. Notice the jump tables which are not states, but a redirection of flow. The Turbo9 state diagram is much smaller and more streamlined than the multicycle version. The multicycle version implementation would require fifteen states

whereas the Turbo9 requires four micro-ops. This demonstrates how much of the sequential logic has been reduced using the decode tables. This should result in the requirement of less transistors and therefore area required to implement the decode stage. The second improvement is the cycles per instruction performance. The pipelined Turbo9 does not require Fetch and Decode micro-ops, so it has a two cycle advantage over the multicycle design. Additionally, for immediate addressing, the instruction can be mapped directly to a micro-op, therefore needing only one cycle.

Table 3-3. Cycles per instruction: Turbo9R vs example multicycle 6809 vs Motorola 6809

Instruction	Instruction Length	Turbo9R	Example Multicycle 6809	Motorola 6809
ADDA Immediate	2	1	4	2
ADDA Direct	2	2	4	4
ADDA Indexed/Indirect	2 to 4	2 to 3	4 to 5	4 to 12
ADDA Extended	3	2	4	5
ADDB Immediate	2	1	4	2
ADDB Direct	2	2	4	4
ADDB Indexed/Indirect	2 to 4	2 to 3	4 to 5	4 to 12
ADDB Extended	3	2	4	5
ADDD Immediate	3	1	4	4
ADDD Direct	2	2	4	6
ADDD Indexed/Indirect	2 to 4	2 to 3	4 to 5	6 to 14
ADDD Extended	3	2	4	7
Range for all instruction variations:		1 to 3	4 to 5	2 to 14
+1 fetch cycle for instr length > 2 bytes		+1	+1	N/A
		1 to 4 cycles	4 to 6 cycles	2 to 14 cycles

Note: The Turbo9R and example multicycle 6809 are both equipped with 16-bit address and data ALUs. They both have a memory bus capable of moving 16-bit unaligned data with no additional latency. The Motorola 6809 cycle data was obtained from the Motorola Programming Manual [4].

Using these two state diagrams we can determine how many cycles the Turbo9 would take vs. our traditional multicycle 6809. We present this data in Table 3-3, and also include the cycle per instruction data for the Motorola 6809. For this example, we used the Turbo9R variant with its memory bus capable of moving unaligned 16-bit data with no additional latency. The multicycle 6809 will have the same memory bus performance capabilities as well as the same 16-bit address and data ALUs. Given this setup, the Turbo9 can execute all the variations of the ADD instruction from 1 to 4 cycles. The multicycle 6809 can execute these instructions in 4 to 6 cycles and the original Motorola implementation takes 2 to 14 cycles.

Summary

In this chapter we have explained how the decode stage of the Turbo9 was designed to give it the performance equal to or better than RISC competitors in the same class. The elegance and orthogonality of the 6809 ISA allowed us to build a CISC to RISC translation layer that is compact and efficient. We introduced microprogramming techniques to the reader, and demonstrated how understanding and reapplying this abstraction layer to a hardwired pipelined decode stage is accomplished. We presented the new and unique μRTL assembler and methodology to support this development. Most literature usually only exposes the reader to a multicycle CISC or a pipelined RISC. Hopefully, this hybrid CISC to RISC microarchitecture enlightens the reader's understanding of advanced microprocessor design techniques.

CHAPTER 4
ANALYSIS

This chapter provides an analysis of the Turbo9 microarchitecture. We will be measuring the performance using benchmarking. We will also present and review FPGA implementation resource usage and relate it to area & power.

Comparison Architectures

In order to provide a reference, we chose several modern processors to measure against. We settled on the following modern microprocessor architectures:

- Motorola 6809 – The original multi-cycle implementation of the ISA
- Atmel AVR – A common 8-bit RISC architecture
- PicoRV32 – A common 32-bit RISC architecture

The Motorola 6809 provides a reference against the original implementation of the ISA. The Atmel AVR is arguably the equivalent 8/16-bit RISC solution for a compact and efficient microprocessor IP [9]. The RISC-V architecture [10] is very prevalent today and the PicoRV32 [11] is a popular scaled down 32-bit microprocessor based on this instruction set.

We used the avr-gcc compiler [12] with an Atmega1284 for the Atmel AVR performance benchmarks. The PicoRV32 Dhrystone results were taken from three sources, two of which were confirmed to use the same speed grade of Xlinix Atrix7 FPGA part [13][14]. The riscv-gcc compiler was used for RISC-V code size numbers [15].

Performance

In order to measure performance, we will select several benchmarks and measure the clock for clock performance, the absolute performance at maximum frequency, and the code size necessary to compile these benchmarks.

Benchmarks

In order to assess the performance of the Turbo9, we chose a number of standard benchmark programs. These include:

Dhrystone. This is a popular synthetic computing benchmark program written in C and developed in 1984 by Reinhold P. Weicke. Dhrystone is representative of system (integer) programming and general microprocessor performance and exercises multiplication and division [16]. It also measures compiler performance. "DMIPS" is a common CPU performance metric and is calculated by dividing dhrystone/sec by 1757, which is the score obtained on aVAX 11/780, a 1 MIPS machine.

BYTE Sieve (C). This C program implements the Sieve of Eratosthenes Algorithm from BYTE Magazine, September 1981 [17]. It is a prime number search algorithm and a good benchmark for exercising memory addressing modes.

BYTE Sieve (assembly). In addition to the BYTE Sieve C-based benchmark program, we hand-coded an assembly version of BYTE Sieve for the Turbo9, Atmel AVR, and RISC-V. We chose this approach to remove the compiler dependency and to allow for hand-optimization opportunities on each microprocessor platform to gauge the best possible performance. The algorithm is easy to understand and scaled well for hand-written assembly language.

Performance Results

The Dhrystone performance per clock in units of DMIPS per MHz is shown in Figure 4-1. This allows a comparison of only the logical design of the microarchitecture and removes frequency scaling benefits of more advanced fabrication processes. The 16-bit Turbo9R is 33% faster than the PicoRV32 RISC-V implementation and 68% faster than the Atmel AVR 8-bit RISC implementation. The regular version of the Turbo9 delivers impressive results given it only

has 8-bits of memory bandwidth. It outperforms the AVR with its 24-bits of memory bandwidth, and the PicoRV32 with its 32-bits of memory bandwidth.

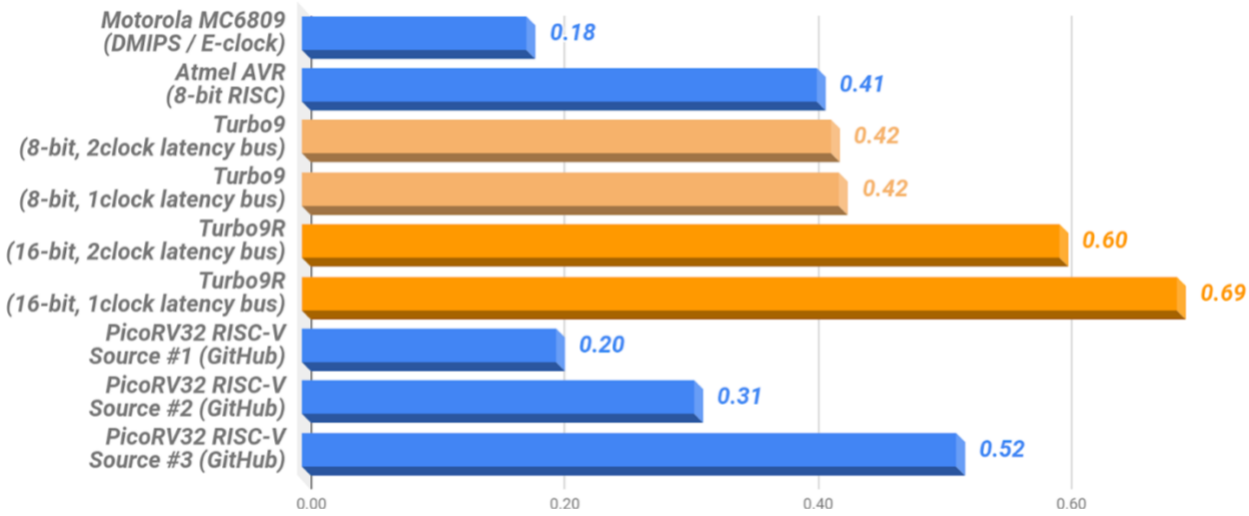


Figure 4-1. Dhrystone performance per clock (DMIPS/MHz, higher is better)

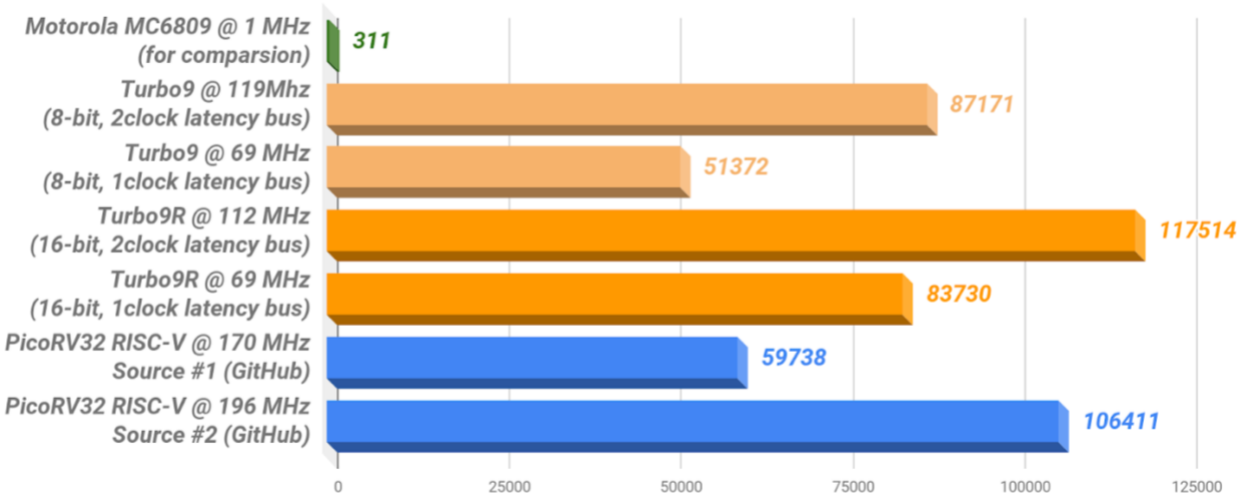


Figure 4-2. Dhrystone FMAX performance in Xilinx Atrix7 FPGA (speed -1) (dhrystones/sec, higher is better)

Figure 4-2 shows the Dhrystone performance in raw Dhrystones per second. In these results, the maximum frequency achievable by the synthesized design is considered. All of the presented microprocessor IPs were synthesized in the same Xilinx Atrix 7 FPGA in order to remove the process frequency scaling factor from the equation. The Turbo9R at its maximum

frequency of 112MHz outperforms the PicoRV32 at its maximum frequency of 196MHz. Also note the original Motorola 6809 implementation is shown here for comparison. A 112MHz Turbo9R is equivalent in performance to a 378MHz Motorola 6809.

The results for the BYTE Sieve benchmark are shown in Figure 4-3. Both the C and assembly version are shown. The Atmel AVR does beat the Turbo9 in the C version by 30%, but the Turbo9 beats the AVR by 13% when comparing the assembly version of the benchmark. This result is due to the excellent and mature GCC compiler used for the AVR. It's a good example of how compiler optimization has a large impact on performance. There could be opportunity here to improve vbcc optimization.

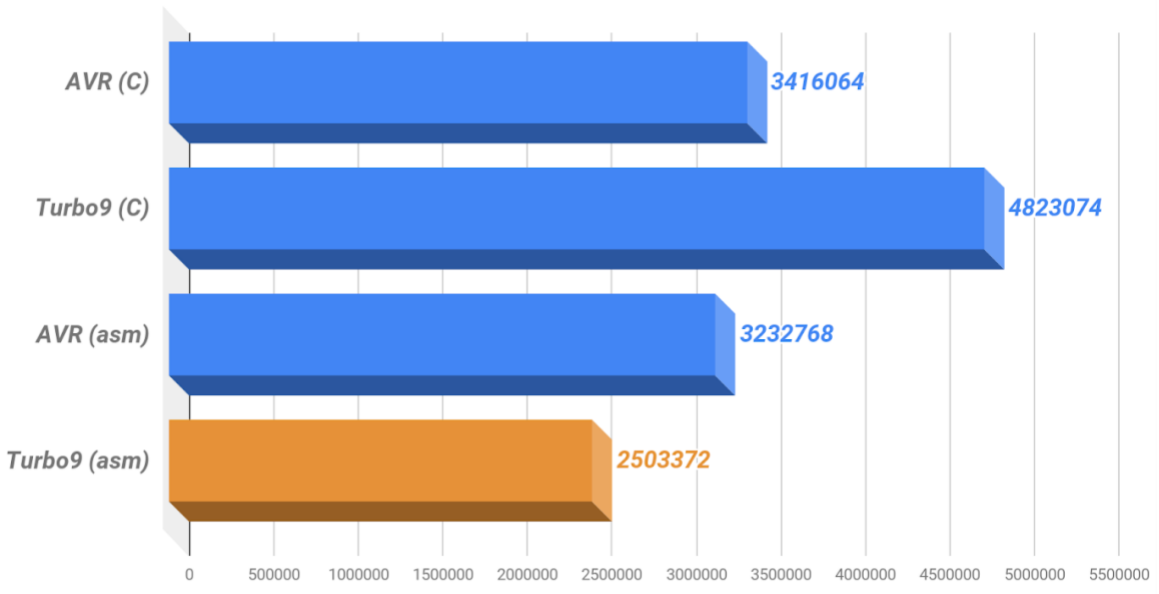


Figure 4-3. BYTE Sieve performance – number of clock cycles (lower is better)

Code Size Results

Code size is important for a compact microprocessor implementation. Smaller memory requirements result in smaller area and lower leakage power. Figure 4-4 shows the code size of Dhrystone compiled for the Turbo9 using different C compilers, along with AVR and RISC-V. All C code used performance rather than size optimization to be representative of the benchmark

results given. The Turbo9 binary was compiled with two optimization levels, both of which outperformed the competing architectures. The O2 optimization level resulted in the smallest code size and still outperformed the AVR and RISC-V. The vbcc-turbo9 binary was smaller than the AVR and RISC-V binaries. The RISC-V 32-bit GCC version is 94 % larger than vbcc-turbo9, and AVR GCC is 5.4% larger than vbcc-turbo9. Also note the vbcc compiler outperformed both GCC and CMOC in code size.

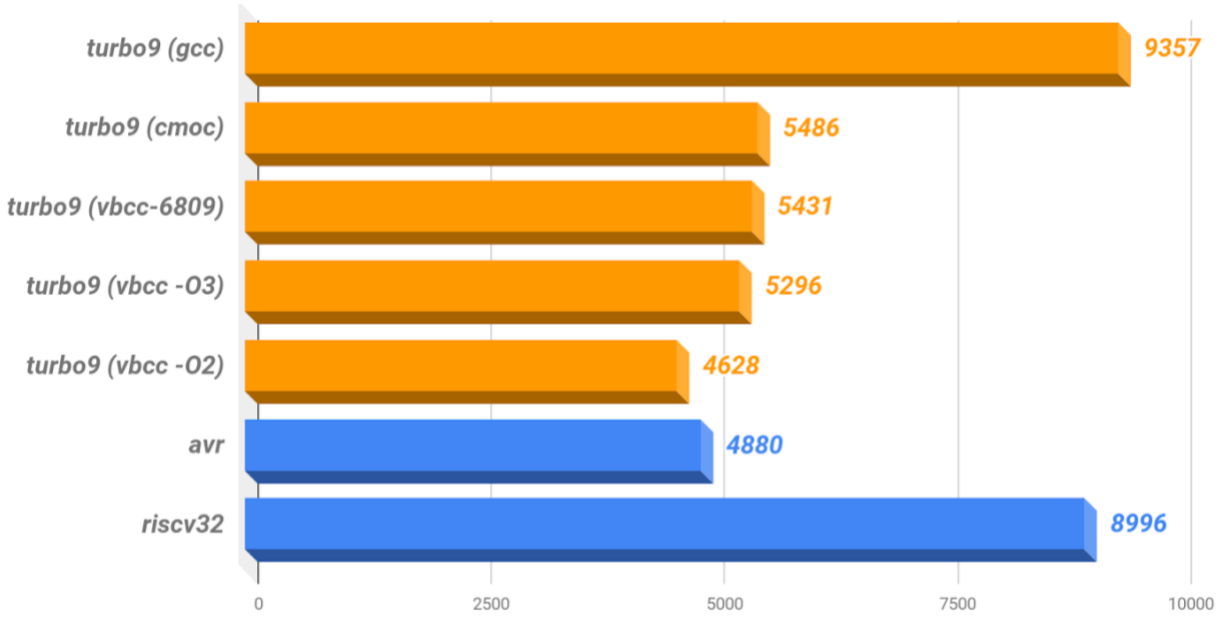


Figure 4-4. Dhrystone – code size in bytes (lower is better)

Figure 4-5 shows the code size for BYTE Sieve in C. Again, the binary emitted by vbcc shows a marked decrease in its size versus the other two 6809 C compilers (GCC and CMOC). Notably, vbcc for the Turbo9 also beat the AVR GCC compiler by 23% and the RISC-V GCC compiler by 91%. The larger sizes of the GCC binaries for the Turbo9 and RISC-V are attributable to the additional library code that the linker pulls in.

The assembly implementation of the BYTE Sieve algorithm focused on the Turbo9, AVR, and RISC-V platforms with the goal of authoring the absolute tightest assembly language versions for each. The work was split amongst team members who advocated strongly for their

chosen architecture. Once the assembly language code was written for each platform, other team members cross-checked the work in order to improve on efficiency where possible.

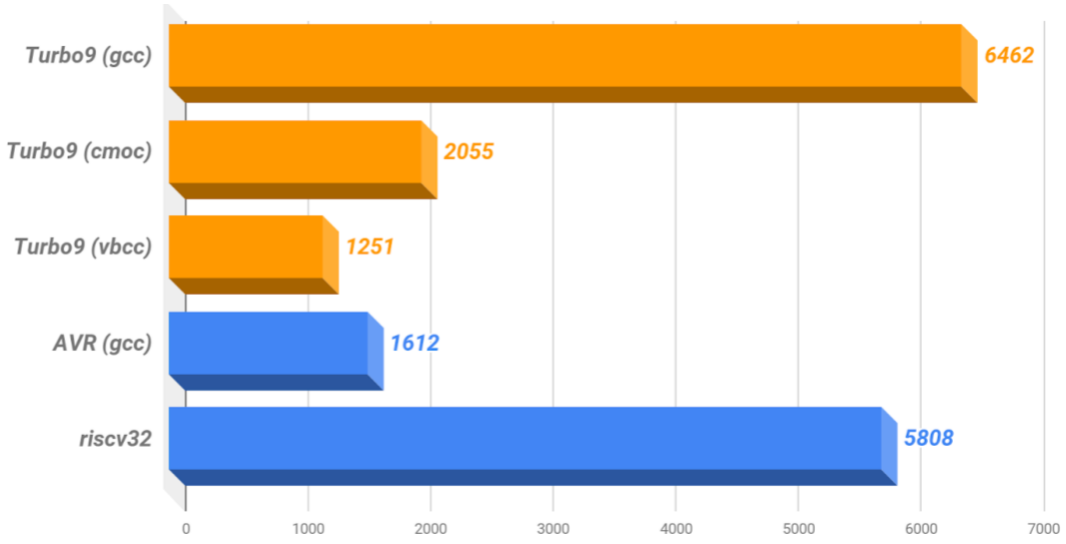


Figure 4-5. BYTE Sieve C – code size in bytes (lower is better)

Figure 4-6 shows that with this fair and balanced approach, the Turbo9 was able to best the AVR by 18% in code size and the RISC-V by 20%. This comparison is notable because it removes the compiler from the equation, allowing for direct optimization assembly techniques.

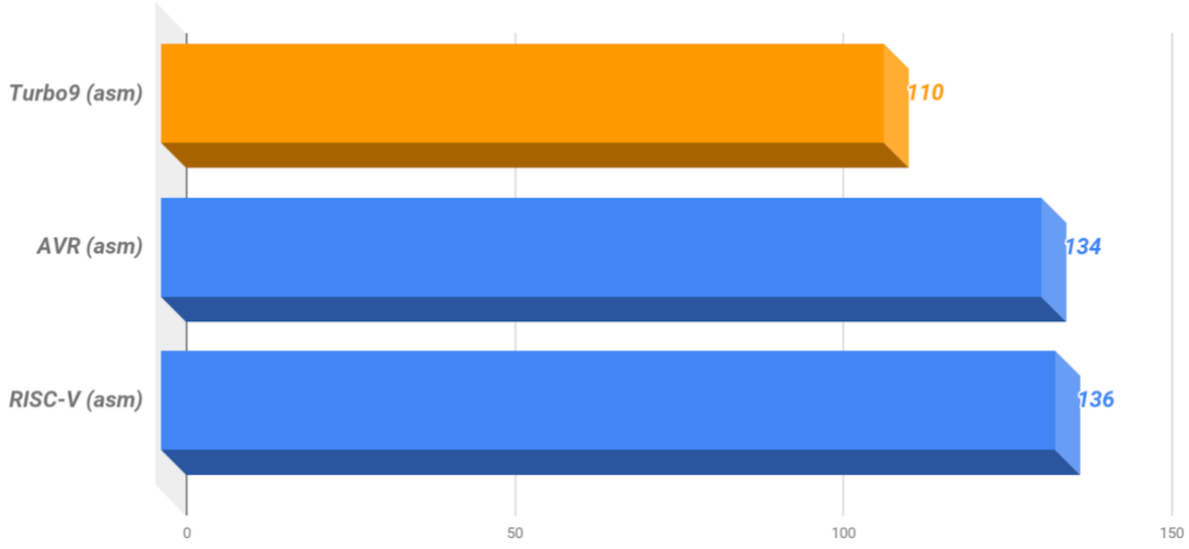


Figure 4-6. BYTE Sieve assembly – code size in bytes (lower is better)

Area & Power

At the time of publishing this paper, we only have FPGA resource utilization data for area analysis. FPGA resources consist of logic look-up tables (LUTs) and flip-flops. There is no accurate way of converting LUTs and flip-flops to the universally accepted two-input NAND gate count. In the future we will be using Yosys, a synthesis tool that will be able to output the gate equivalate data.

Table 4-1. Turbo9 and PicoRV32 FPGA resource usage

	PicoRV32(regular)	PicoRV32(large)	Turbo9	Turbo9R
DMIPS/MHz	0.20	0.52	0.42	0.69
6 inputs LUTs	1000	2107	1227	1464
Flip Flops	701	1085	455	505

Table 4-1 shows the FPGA resource usage for two variations of the Turbo9 and two variations of the PicoRV32 along with their performance rating in DMIPS per MHz. Note both Turbo9 variations use significantly less flip-flops than the PicoRV32. This is due to the small register set of the 6809 programming model compared to the RISC-V's large number of registers. The Turbo9's LUT count is slightly higher than the smaller version of the PicoRV32, though its performance is 2X to 3.5X faster. Since the area of a flop should be greater than the logic equivalent of one LUT, we expect that the equivalent gate count comparison would show that the Turbo9 is smaller overall for all cases. We will verify this claim when we run the Yosys logic synthesis tool. Also note there is still a large potential for area optimization for the Turbo9. One area specifically is to remove the combinatorial decoding of the pre-byte, and to sacrifice one clock cycle for opcodes that occur on page 2 or 3 of the opcode matrix. Doing so should have a small impact on performance, but would gain a large reduction in area by removing four 8-bit registers from the instruction queue and a fair amount of logic from the instruction pre-decoder.

Regarding power, it can be shown that active power is proportional to the switching capacitance times the frequency if supply voltage is held constant. The switching capacitance is proportional to the number of transistor gates being charged and discharged in a CMOS device. Therefore, the capacitance is proportional the gate equivalent of a design. The frequency can be related to the DMIPS/MHz performance rating. The higher the DMIPS/MHz rating, the lower the frequency required to meet the desired performance level. Therefore, the smaller the microprocessor with a higher DMIPS/MHz rating will achieve lower active power. Leakage power is also directly proportional to area, given that it is proportional to the number of transistors. We will also be measuring power directly in the future to verify the Turbo9's low power usage.

CHAPTER 5
TURBO9 TOOLKIT

As we've previously stated, our goal is to create a professional level IP. A collection of hardware and software development tools are expected of a commercial IP such as ARM. During the development of the Turbo9 we have drawn from an extensive library of 6809 tools, and we have developed some of our own tools.

IP Development Tools

This is an open-source IP, therefore it is expected that a set of development tools is available to the consumer for implementing or even customizing the Turbo9 IP.

Verilog Tools

For Verilog RTL development and behavioral simulation we began our development using Cadence tools provided by the University of Florida's Electrical and Computer Engineering Department. We soon came to realize not everyone has access to these expensive software licenses, and since this is an open-source IP it should use open-source tools. Therefore, early in the project we ported our code base to Icarus Verilog [18] which is an excellent open-source Verilog 2001 simulator. All of our run and regression scripts are built around Icarus Verilog. In addition, we use GTKwave [19] to view waveforms produced by our verification test bench.

Microcode Tools

We developed μRTL, which is a custom microcode assembler for developing high-performance decode stages. Instead of outputting a microcode ROM, it produces optimized and efficient RTL blocks to be synthesized into gates. There is a focus on balancing parallel versus sequential decoding. This tool is covered in-depth in Chapter 3.

Synthesis Tools

All of our development up to this point has utilized an FPGA target platform. We use the Digilent Arty A7 development board that contains a Xilinx Atrix7 FPGA with a -1 speed grade [20]. To synthesize and configure this target FPGA, we use the Xilinx Vivado software [21]. We chose this specific FPGA because it is a modern part that is supported by the free version of the Xilinx software. We plan on using the open-source synthesis tool, Yosys, to synthesize for a standard cell library in the future [22].

Verification Testbench

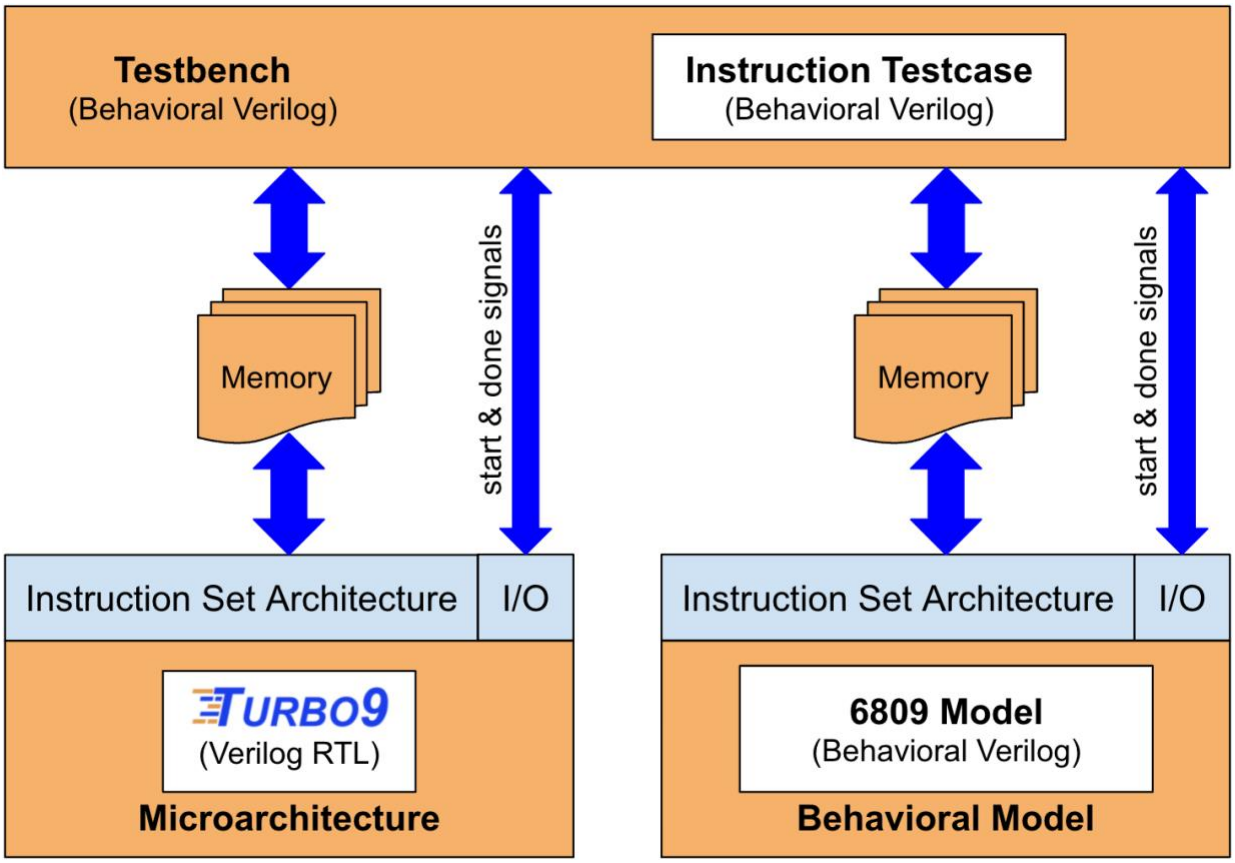


Figure 5-1. Turbo9 verification testbench block diagram

A full verification test bench is expected from commercial IP vendors such as ARM. This is not merely a test bench for simple simulation. This is a full self-checking verification suite that has full ISA coverage, covers over 4500 attributes, with randomized and directed testcases.

The Turbo9 Verification test bench block diagram is shown in Figure 5-1. A complete model of the 6809 instruction set, plus Turbo9 extensions, was written in behavioral Verilog. This was then encapsulated in a top-level testbench along with the Turbo9 RTL. Thirty four separate test cases produce customized machine language test programs with directed and randomized inputs for the instruction under test. This test program is then executed on both the Turbo9 RTL and 6809 behavioral model. After each signals the test bench that they have completed the test, the test bench compares the output values in these memory images to verify that they match.

The level of effort necessary to create this verification test bench is equal to the microarchitecture design effort. It is an extremely useful tool during not only the development, but also the optimization phase as it ensures that the Turbo9 always meets the ISA specification. Providing a full verification suite with the Turbo9 microprocessor IP validates its professional level and separates it from amateur projects.

Software Development Tools

The Turbo9 software development toolkit is summarized in Figure 5-2. Note, the four different targets available to the programmer to run and test their code. Using these platforms, the software developer can run their code in real time on hardware, do a simple instruction level simulation in Linux, or an in-depth cycle accurate Verilog simulation.

C Compilers

Software compilation is a huge factor in performance so we want to choose the best optimizing C compiler for the Turbo9. After surveying the available C compilers for the 6809, which the Turbo9 derives from, we identified three candidates:

- GCC - A long-standing C compiler that has a 6809 backend patch available [23].
- CMOC - Created by a 6809 enthusiast and has been available for a few years [24].
- vbcc – An optimizing C compiler that supports several backends including 6809 [25].

In our tests, vbcc outperformed GCC and CMOC in both code size and execution speed for C-based programs on the Turbo9 due to its high degree of optimization. The author of the vbcc compiler has spent considerable time and effort to focus on optimization and has even worked with the Turbo9 team to add support for processor-specific instructions. This degree of cooperation between a compiler author and a microprocessor design team ensures the highest level of performance. Having the support of a compiler author is a huge asset to our project.

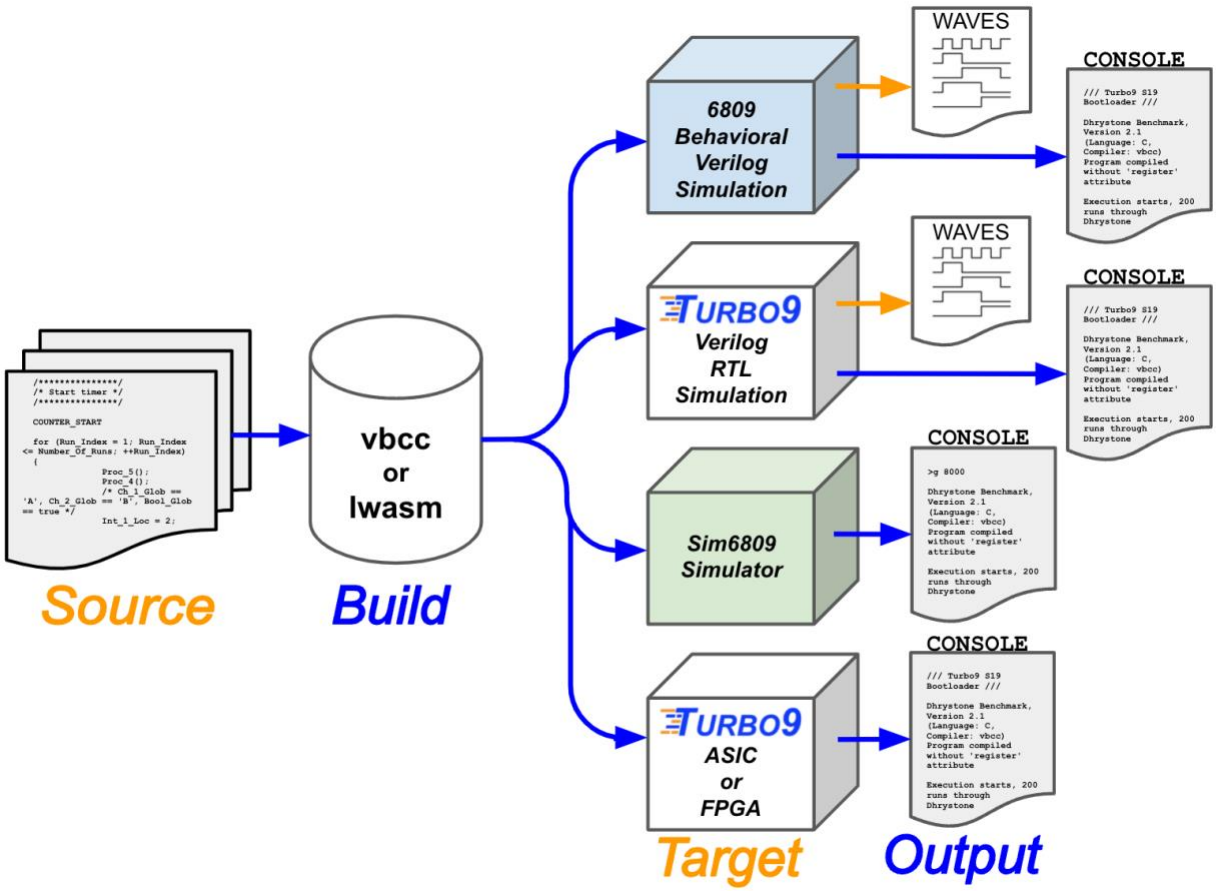


Figure 5-2. Turbo9 software development toolkit

Assemblers

There are many 6809 assemblers available. Our preferred choices are lwasm and vasm. The powerful lwasm cross-assembler is included in lwtools, which also includes a linker, lwlink [26]. The vbcc C compiler uses vasm and vlink.

Simulators

Sim6809 is an open-source 6809 simulator written in C to which we have successfully added Turbo9 extensions [27]. This simulator is very convenient and lightweight to use.

In addition to the verification test bench, a set of simulation tools are available to run Turbo9 code on the Verilog behavioral model, as well as the Verilog RTL. These tools are mature enough now that they also simulate the console output from the asynchronous serial port. The major advantage of using Verilog simulation during software development is the ability to view cycle accurate waveforms of the internals of the Turbo9 hardware.

FPGA or ASIC Platform

As mentioned earlier, we use the Digilent Arty A7 board for FPGA development. Running code on an FPGA gives us real-time performance and allows us to run much more complex software programs faster than simulation. To support these platforms, we have developed a small SoC design that combines the Turbo9 microprocessor with memory, basic I/O, and an asynchronous serial port. We also developed a bootloader to facilitate loading programs into memory. This bootloader is also supported in the Verilog simulators.

CHAPTER 6
CONCLUSION

The Turbo9 is an open-source compact high performance pipelined 16-bit microprocessor IP delivered in a professional package, making it the superior solution versus other 32-bit solutions in certain applications. The benchmarks confirm the expected performance of the Turbo9's microarchitecture. Its performance compared to competing microprocessor IPs is excellent. The core has proven to be compatible with third party software tools through testing with multiple C compilers. The area was measured and shown to be superior.

The next step for the Turbo9 is to improve the performance by adding branch prediction. An improvement in area and power can also be achieved with further microarchitecture optimizations. We will pursue synthesis in a standard cell library as feedback to this optimization effect.

Overall, we have created a useful tool for SoC designers to fulfill the requirements for high-level control in their SoC designs. The Turbo9 is the microprocessor solution for compact, high performance applications where size and efficiency matter.

APPENDIX A
TURBO9 OPCODE MATRIX

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

Page 2 $10	Most Significant Nibble															
	DIR		REL		ACCA	ACCB	IDX / IND	EXT	IMM	DIR	IDX / IND	EXT	IMM	DIR	IDX / IND	EXT
Least Significant Nibble	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0			LBRA													
1			LBRN													
2			LBHI													
3			LBLS						CMPD							
4		EDIV	LBHS,LBCC													
5		EDIVS	LBLO,LBCS													
6			LBNE													
7			LBEQ													
8		IDIVS	LBVC													
9		FDIV	LBVS													
A			LBPL													
B			LBMI													
C			LBGE						CMPY							
D			LBLT													
E			LBGT						LDY				LDS			
F			LBLE	SWI2						STY				STS		

Page 3 $11	Most Significant Nibble															
	DIR		REL		ACCA	ACCB	IDX / IND	EXT	IMM	DIR	IDX / IND	EXT	IMM	DIR	IDX / IND	EXT
Least Significant Nibble	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0																
1																
2																
3									CMPU							
4																
5																
6																
7																
8																
9																
A																
B																
C									CMPS							
D																
E																
F				SWI3												

Figure A-1. Turbo9 opcode matrix

Indexed Address Postbyte			Most Significant Nibble															
			X	X	Y	Y	U	U	S	S	X	X	Y	Y	U	U	S	S
			0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
			0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Least Significant Nibble	0000	0	0,X	-16,X	0,Y	-16,Y	0,U	-16,U	0,S	-16,S	,X+	[,X+]	,Y+	[,Y+]	,U+	[,U+]	,S+	[,S+]
	0001	1	1,X	-15,X	1,Y	-15,Y	1,U	-15,U	1,S	-15,S	,X++	[,X++]	,Y++	[,Y++]	,U++	[,U++]	,S++	[,S++]
	0010	2	2,X	-14,X	2,Y	-14,Y	2,U	-14,U	2,S	-14,S	,-X	[,-X]	,-Y	[,-Y]	,-U	[,-U]	,-S	[,-S]
	0011	3	3,X	-13,X	3,Y	-13,Y	3,U	-13,U	3,S	-13,S	,--X	[,--X]	,--Y	[,--Y]	,--U	[,--U]	,--S	[,--S]
	0100	4	4,X	-12,X	4,Y	-12,Y	4,U	-12,U	4,S	-12,S	,X	[,X]	,Y	[,Y]	,U	[,U]	,S	[,S]
	0101	5	5,X	-11,X	5,Y	-11,Y	5,U	-11,U	5,S	-11,S	B,X	[B,X]	B,Y	[B,Y]	B,U	[B,U]	B,S	[B,S]
	0110	6	6,X	-10,X	6,Y	-10,Y	6,U	-10,U	6,S	-10,S	A,X	[A,X]	A,Y	[A,Y]	A,U	[A,U]	A,S	[A,S]
	0111	7	7,X	-9,X	7,Y	-9,Y	7,U	-9,U	7,S	-9,S	D,X	[D,X]	D,Y	[D,Y]	D,U	[D,U]	D,S	[D,S]
	1000	8	8,X	-8,X	8,Y	-8,Y	8,U	-8,U	8,S	-8,S	8bit,X	[8bit,X]	8bit,Y	[8bit,Y]	8bit,U	[8bit,U]	8bit,S	[8bit,S]
	1001	9	9,X	-7,X	9,Y	-7,Y	9,U	-7,U	9,S	-7,S	16bit,X	[16bit,X]	16bit,Y	[16bit,Y]	16bit,U	[16bit,U]	16bit,S	[16bit,S]
	1010	A	10,X	-6,X	10,Y	-6,Y	10,U	-6,U	10,S	-6,S	D,X	[D,X]	D,Y	[D,Y]	D,U	[D,U]	D,S	[D,S]
	1011	B	11,X	-5,X	11,Y	-5,Y	11,U	-5,U	11,S	-5,S	D,X	[D,X]	D,Y	[D,Y]	D,U	[D,U]	D,S	[D,S]
	1100	C	12,X	-4,X	12,Y	-4,Y	12,U	-4,U	12,S	-4,S	8bit,PC	[8bit,PC]	8bit,PC	[8bit,PC]	8bit,PC	[8bit,PC]	8bit,PC	[8bit,PC]
	1101	D	13,X	-3,X	13,Y	-3,Y	13,U	-3,U	13,S	-3,S	16bit,PC	[16bit,PC]	16bit,PC	[16bit,PC]	16bit,PC	[16bit,PC]	16bit,PC	[16bit,PC]
	1110	E	14,X	-2,X	14,Y	-2,Y	14,U	-2,U	14,S	-2,S	8bit	[8bit]	8bit	[8bit]	8bit	[8bit]	8bit	[8bit]
	1111	F	15,X	-1,X	15,Y	-1,Y	15,U	-1,U	15,S	-1,S	16bit	[16bit]	16bit	[16bit]	16bit	[16bit]	16bit	[16bit]

Figure A-2. Turbo9 postbyte matrix for indexing addressing modes

APPENDIX B
µRTL DECODE BLOCK MAPPING & STATISTICS

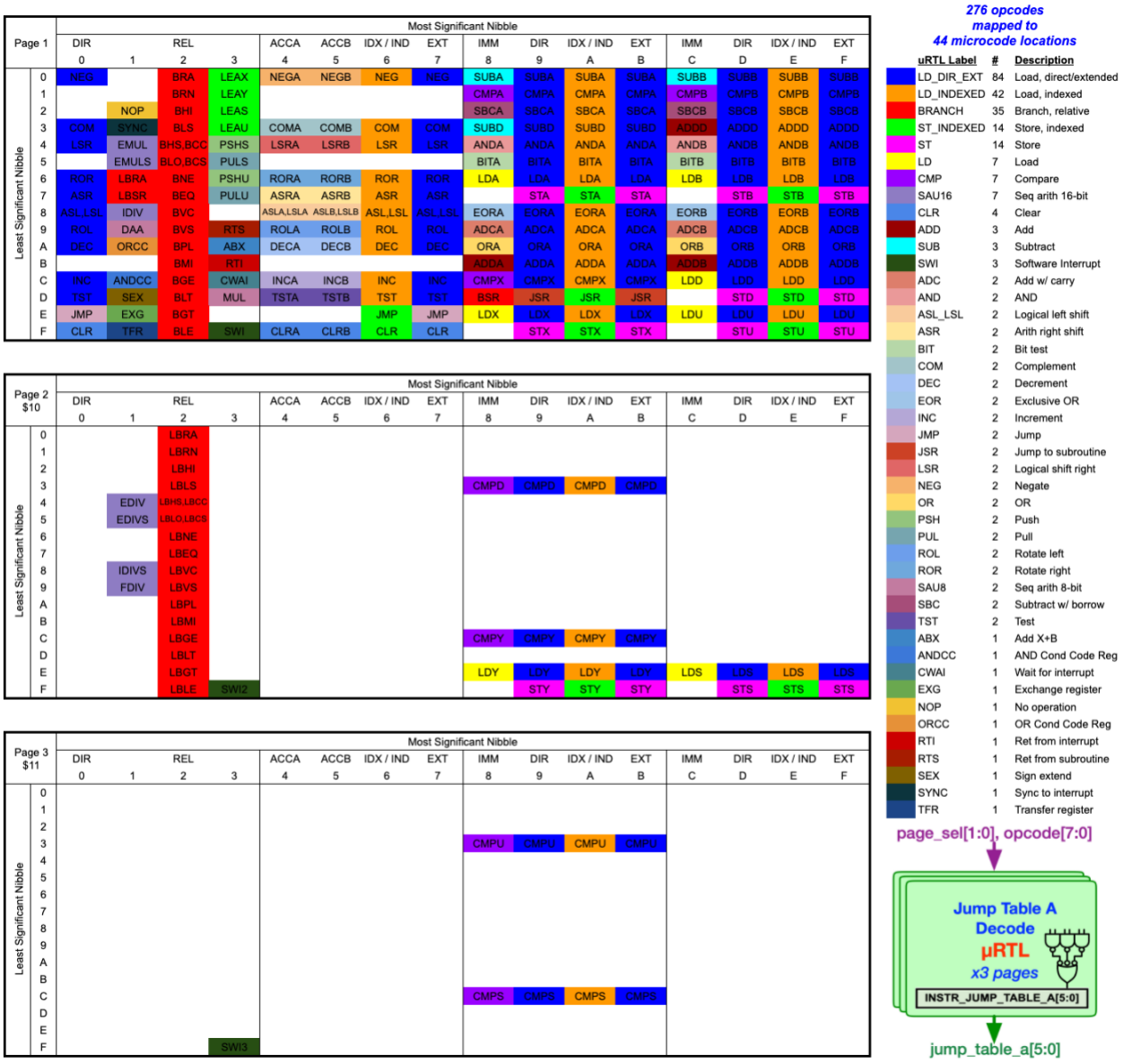


Figure B-1. Jump table A decode block generated by µRTL

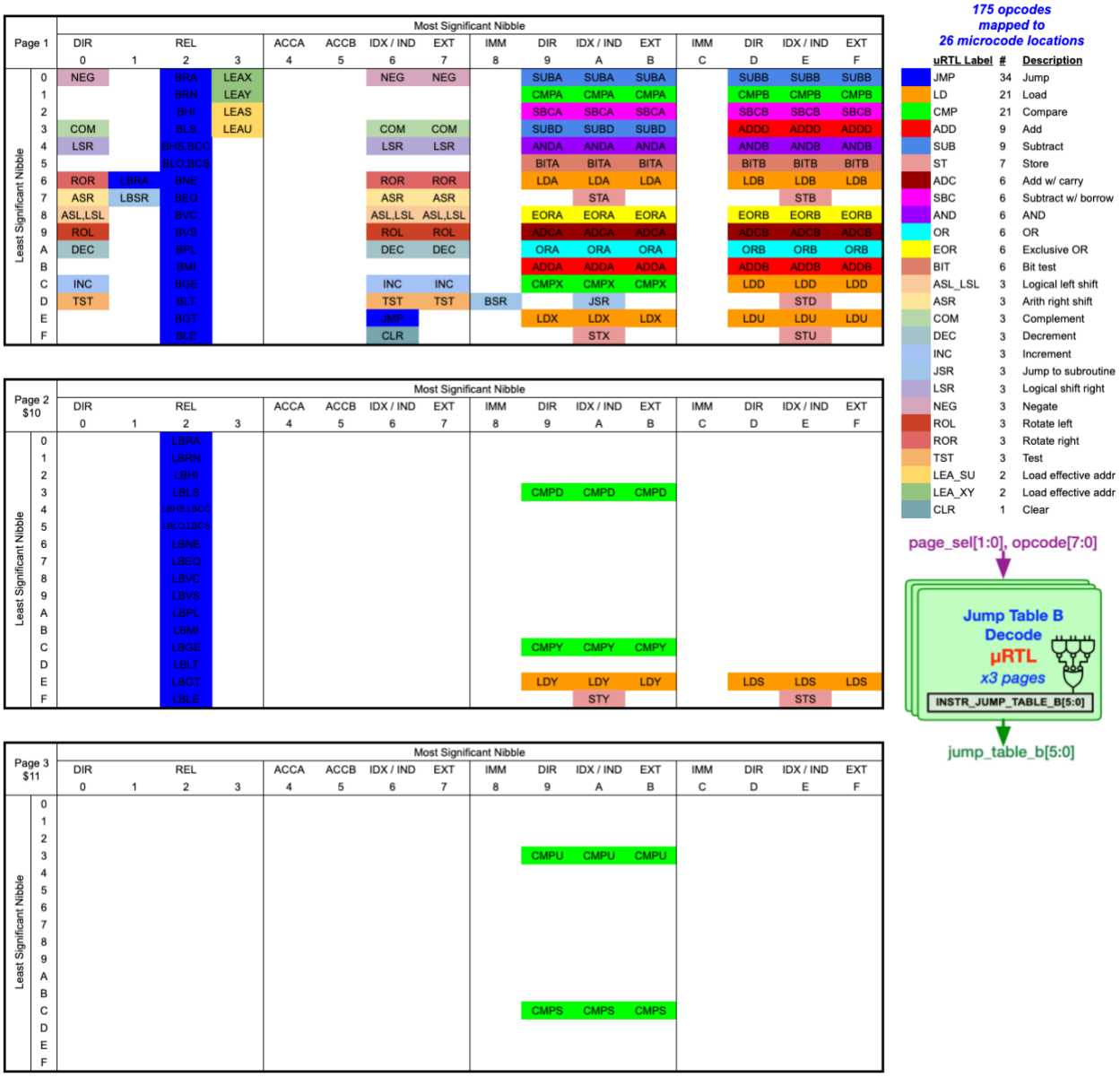


Figure B-2. Jump bable B decode block generated by µRTL

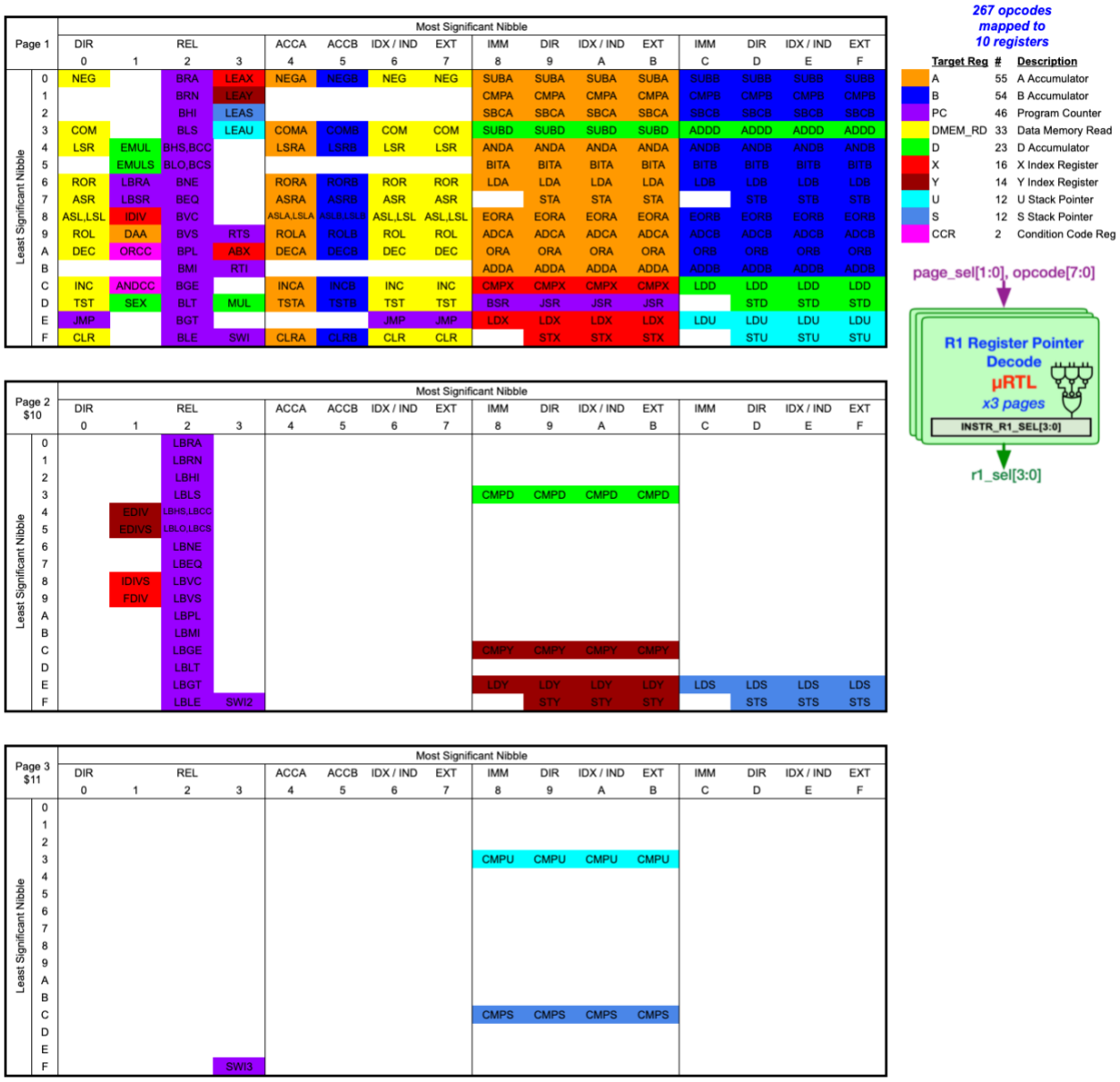


Figure B-3. R1 register pointer decode block generated by µRTL

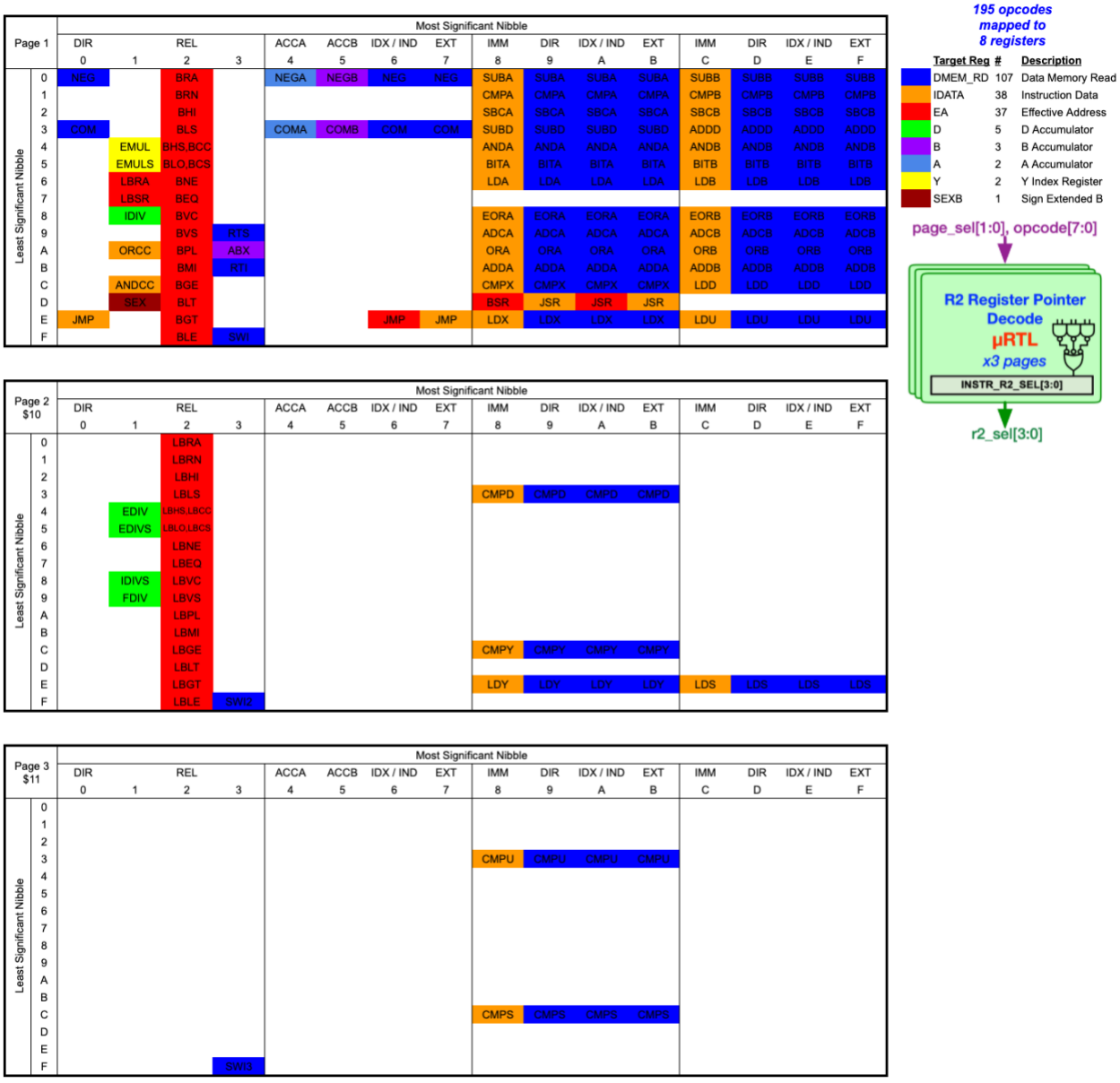


Figure B-4. R2 register pointer decode block generated by µRTL

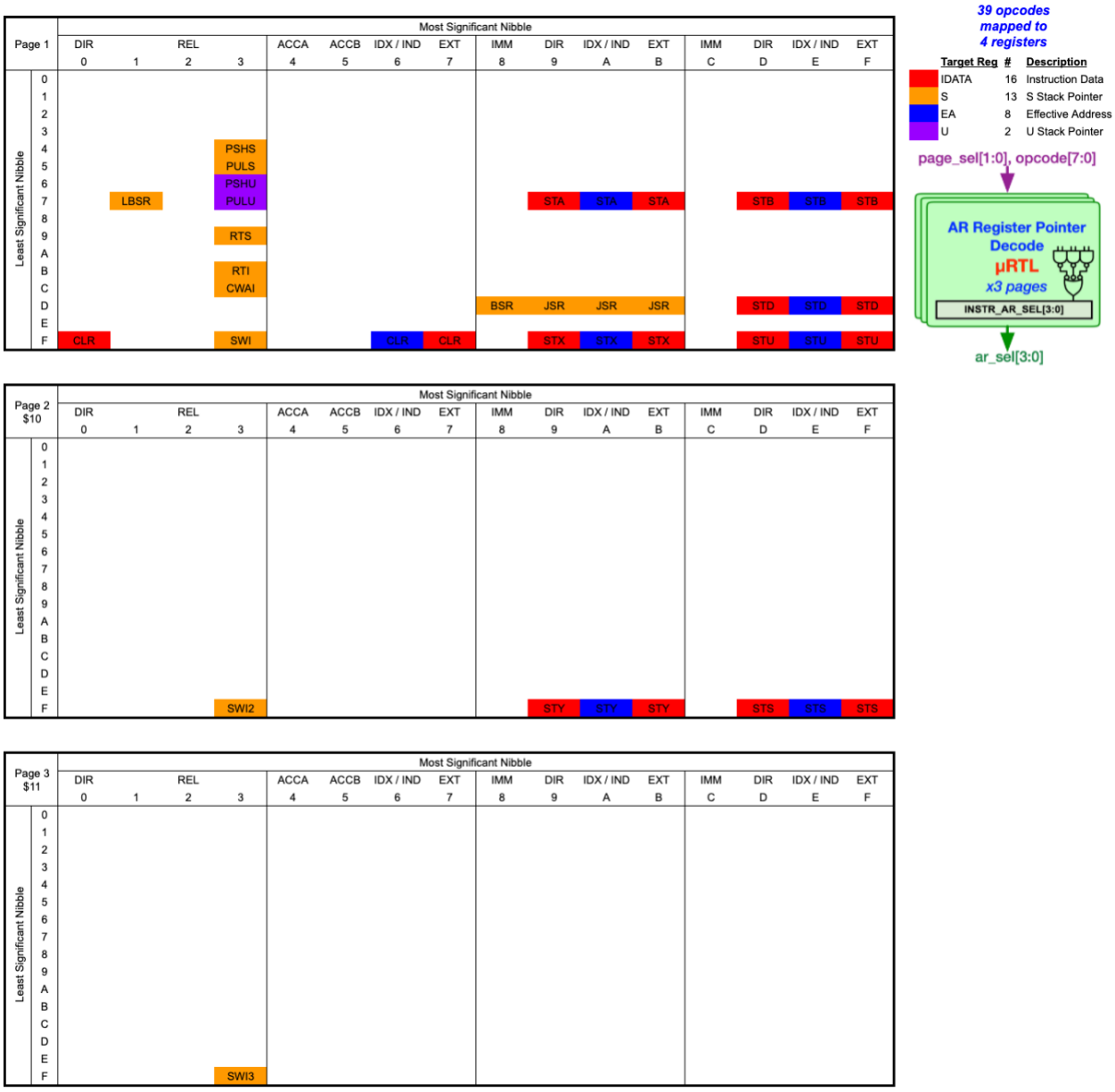


Figure B-5. AR register pointer decode block generated by µRTL

LIST OF REFERENCES

[1] K. Phillipson, M. Rywalt and B. Pitre, “Turbo9 Microprocessor IP,” March 27, 2022. [Online]. Available: https://github.com/turbo9team/turbo9 [Accessed on: March 27, 2022]

[2] D. A. Patterson, J. L. Hennessy, *Computer Organization & Design: The Hardware / Software Interface,* San Francisco, CA: Morgan Kaufmann Publishers, Inc., 1994

[3] T. Ritter and J. Boney, “A Microprocessor for the Revolution: The 6809,” *BYTE magazine*, Jan-Feb 1979

[4] Motorola, *MC6809-MC6809E Microprocessor Programming Manual*, Motorola Inc., 1981

[5] OpenCores, *Wishbone B4: WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores*, OpenCores, 2010

[6] Rohita P. Patil and Pratima V. Sangamkar, “A Review of System-On-Chip Bus Protocols,” *International Journal of Advanced Research in Electrical, Electronics and Instrumentation Engineering*, Vol. 4, Issue 1, January 2015

[7] M. A. Lynch, *Microprogrammed State Machine Design*, Boca Raton, FL: CRC Press, Inc., 1993

[8] M. Smotherman, “A Brief History of Microprogramming”, Clemson University, July 2019. [Online]. Available: https://people.cs.clemson.edu/~mark/uprog.html [Accessed on: April 23, 2020]

[9] Atmel, *AVR Instruction Set Manual*, Atmel Corporation, 2016

[10] RISC-V International, “RISC-V,” 2021. [Online]. Available: https://riscv.org/ [Accessed on: March 27, 2022]

[11] YosysHQ, “PicoRV32 – A Size-Optimized RISC-V CPU,” March 2, 2019. [Online]. Available: https://github.com/YosysHQ/picorv32 [Accessed on: March 27, 2022]

[12] Free Software Foundation, “avr-gcc – GCC Wiki,” Jun 27, 2021. [Online]. Available: https://gcc.gnu.org/wiki/avr-gcc [Accessed on: March 27, 2022]

[13] Rik te Winkel, “Softcore Comparasions,” June 21, 2020. [Online]. Available: https://github.com/riktw/SoftcoreComparisons [Accessed on: March 27, 2022]

[14] Kammoh, “PicoRV32 - A Size-Optimized RISC-V CPU,” 2021. [Online]. Available: https://githubmemory.com/repo/kammoh/picorv32 [Accessed on: March 27, 2022]

[15] RISC-V Software Collaboration, “RISC-V GNU Compiler Toolchain,” March 27, 2022. [Online]. Available: https://github.com/riscv-collab/riscv-gnu-toolchain [Accessed on: March 27, 2022]

[16] R. P. Weicker, “Dhrystone Benchmark: Rationale for Version 2 and Measurement Rules,” *SIGPLAN Notices*, Vol 23, Issue 8, Aug 1988

[17] J. Gilbreath and G. Gilbreath, “Eratosthenes Revisted: Once More through the Sieve,” *BYTE magazine*, Jan 1983

[18] S. Williams, “Icarus Verilog,” March 27, 2022. [Online]. Available: http://iverilog.icarus.com [Accessed on: March 27, 2022]

[19] U. Finkelstein, “GTKwave,” Nov 14, 2020. [Online]. Available: http://gtkwave.sourceforge.net/gtkwave.pdf [Accessed on: March 27, 2022]

[20] Digilent, “Digilent,” March 27, 2022. [Online]. Available: https://digilent.com [Accessed on: March 27, 2022]

[21] Xilinx, “Xilinx Vivado,” March 27, 2022. [Online] Available: https://www.xilinx.com/products/design-tools/vivado.html [Accessed on: March 27, 2022]

[22] Clifford Wolf, “Yosys Open Synthesis Suite,” March 27, 2022. [Online]. Available: http://bygone.clairexen.net/yosys/ [Accessed on: March 27, 2022]

[23] T. Volden, “Motorola 6809 dev tools – gcc-6809,” Nov 11, 2021. [Online] Available: https://launchpad.net/~tormodvolden/+archive/ubuntu/m6809 [Accessed on: March 27, 2022]

[24] P. Sarrazin, “CMOC - 6809 cross-compiler for a C-like language,” March 5, 2022. [Online]. Available: http://perso.b2b2c.ca/~sarrazip/dev/cmoc.html [Accessed on: March 27, 2022]

[25] V. Barthelmann, *Advanced Compiling Techniques to reduce RAM Usage of Static Operating Systems*, Dissertation zur Erlangung des Grades Doktor-Ingenieur, Erlangen, Januar 2004, Available: http://www.compilers.de/publications_eng.html [Accessed on: March 27, 2022]

[26] W. Astle, “LWTOOLS,” Jan 25, 2022. [Online]. Available: http://www.lwtools.ca/ [Accessed on: March 27, 2022]

[27] J. Thoen, “Sim6809,” Jul 18, 2016. [Online]. Available: https://github.com/gordonjcp/sim6809 [Accessed on: March 27, 2022]

BIOGRAPHICAL SKETCH

Kevin Phillipson is a graduate student at the University of Florida and has been working as an ASIC design engineer in industry since 2008. He also completed his Bachelor of Science in Electrical Engineering at the University of Florida in 2008. During his career Kevin has successfully developed many digital and mixed signal designs. His design experience includes simulation, tape-out, test and debug in deep submicron fabrication processes.

Kevin has always had an interest in computer architecture and microprocessor design. This passion has led him to pursue the research for this master's thesis.