**3-Stage Pipelined RV32i CPU for resource-constrained Applications**

# The CPU Core

## Introduction

The report outlines a small footprint pipelined implementation of the RV32i ISA. The implementation targets resources-constrained applications; hence, several design decisions were made to achieve such goal.

The following hard requirements are considered:

1. No separation between data memory and instruction memory. Hence, no L1 caches are needed and the CPU complies to Von Neumann memory architecture
2. The Register File must be implemented using memory dual-port SRAM generated using a memory compiler.
3. Data hazards due to pipelining must be handled by the hardware
4. Control instructions (jump and branch) flush the pipeline

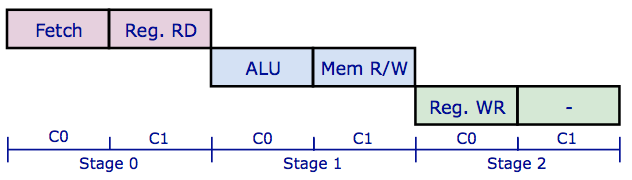


Figure . The CPU Pipeline Stages

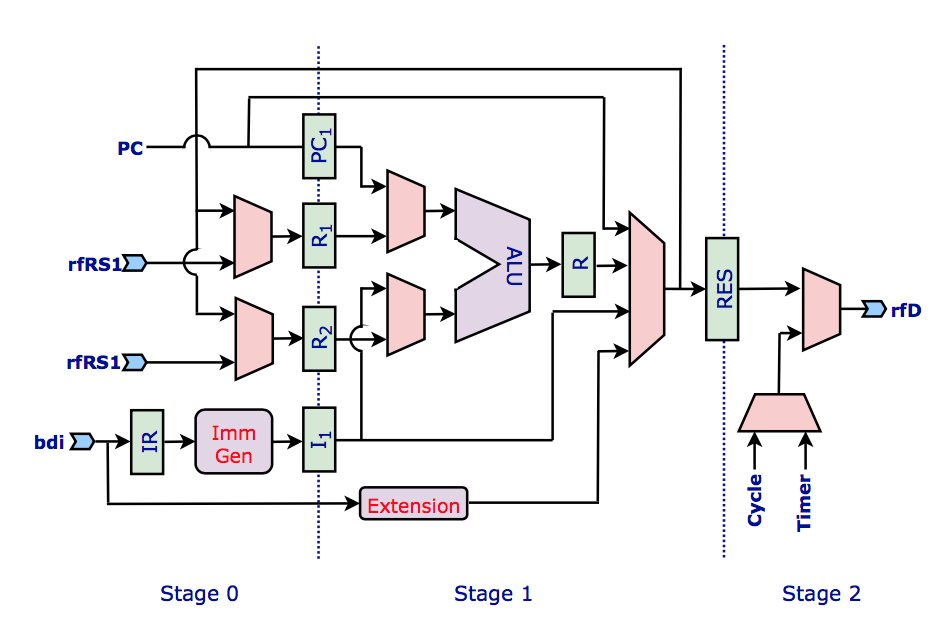
****

Figure . Part of the Datapath showing the Pipeline Stages

## The pipeline

The CPU executes each instruction in 6 cycles divided into 3 stages of 2 cycles each (C0 and C1).

* Stage 0: Instruction Fetch (C0) and Registers read (C1).
* Stage 1: ALU operation (C0) and Memory read/write (C1).
* Stage 2: Register write back (C0)

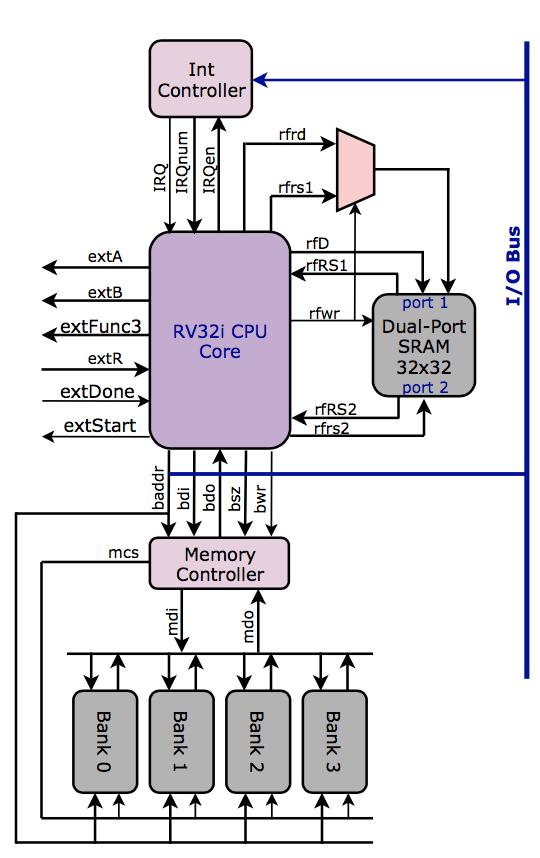


Figure 3. The CPU External Interfaces

## The Register File

The CPU needs dual port SRAM (32 x 32) to act as a register file. By doing so, we avoided the complications associated with designing and implementing a custom register file of 3 ports (2 read ports and 1 write port) as the dual port memory is generated using a memory compiler. However, using a dual port memory for the register file, prevents the datapath of the CPU from reading and writing 3 different registers on one clock cycle. This is solved in the way the datapath pipeline is implemented. Registers reads (2 of them) are done concurrently on one clock cycle (C1 of stage 0) and the register write is done on a separate clock cycle (C0 of stage 2).

To speedup context switching in Real-Time OS (RTOS), a 64 x 32 dual port SRAM might be used to implement 2-bank register file (more details will be added when the RTOS is implemented).

## The CPU-Memory Interface

The CPU is interfaced to the memory and the I/O devices using a 32-bit bus that enables reading or writing 8-bit, 16-bit or 32-bit words in one transaction. To interface this bus to the main memory, 2 things must be considered:

1. The main memory has to be banked (4 x 8-bit banks)
2. A memory controller has to be implemented to enable memory banks to be written and to multiplex the memory banks data output

## Expansion Port

Adding support for new instructions (e.g., multiplication) through non-standard ext. The new instructions have opcode 10\_001. func3 field is used to specify the instruction.

## I/O Devices

I/O devices are connected to same bus as the memory. All I/O registers are memory mapped and are located at location starting from address 0x8000\_0000. The convention is to use 0x8000\_0000 as a starting address for device 0, 0x8100\_0000 as a starting address for device 1, and so on. The CPU has a single IRQ line so an Interrupt Controller must be used to expand this to either 8 or 16 IRQ lines.

## Interrupts

2 instructions are there to help with interrupts:

* Wait for an Interrupt (***wfi***). ***wfi*** instruction is supported and must be followed by 2 ***nop*** instructions!
* URET instruction is implemented to return from an ISR

The core does not support interrupts nesting. Interrupts are disabled during serving the current interrupt. uie register (register 0x4) is used for enabling and disabling interrupts.

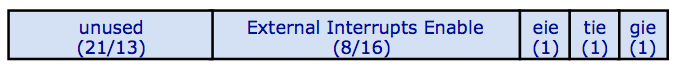


Figure . UIE Register Format

There are 4 interrupt vectors starting from address 16. 4 words are reserved for every ISR (these 4 instructions are good enough to do necessary initializations then jumping to the ISR main code)

|  |  |
| --- | --- |
| **Interrupt Source** | **ISR Starting Address (decimal)** |
| ECALL instruction | 16 |
| EBREAK instruction | 32 |
| Timer | 48 |
| External Interrupt | 64 |

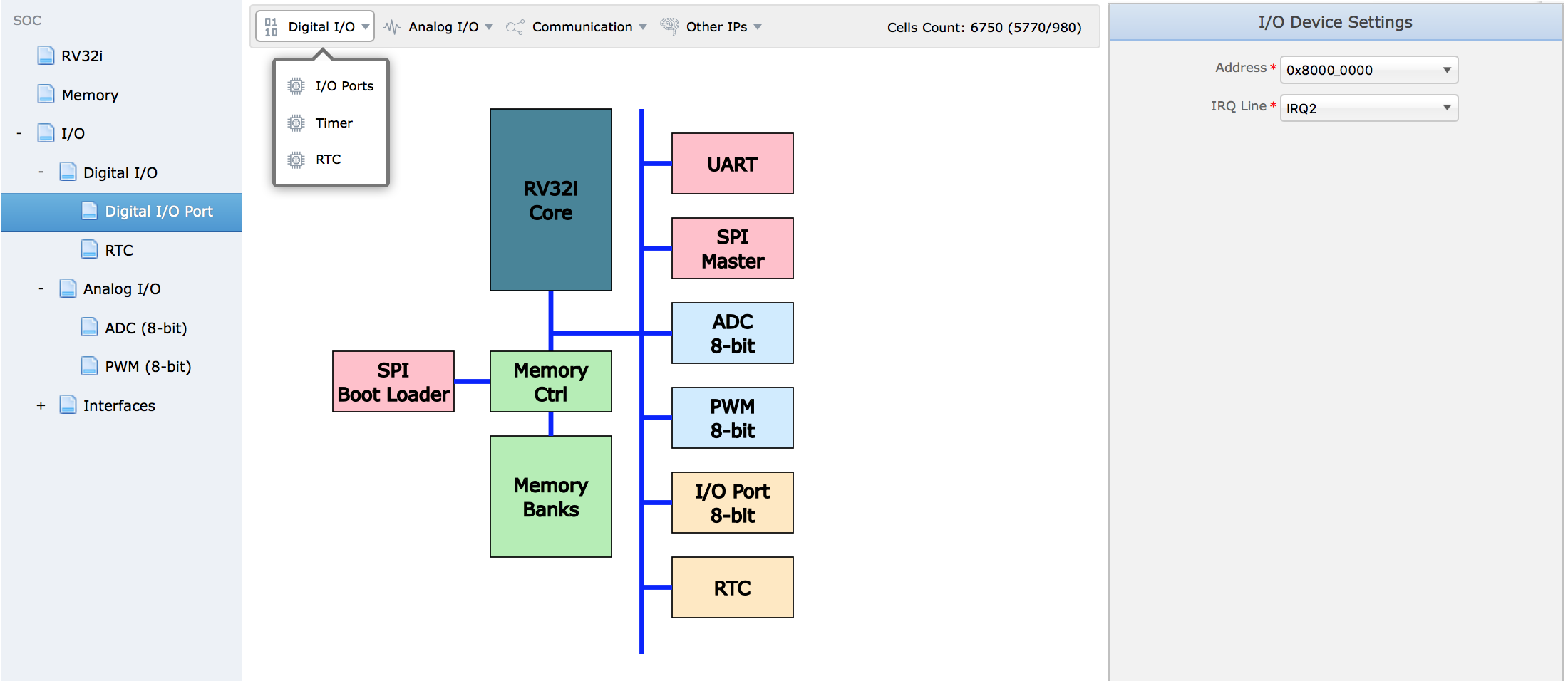
## Counters

Only 32-bit Cycle and Time registers. The Cycle register in incremented every clock cycle. The Timer register is a down counter and it generates an interrupt when it reaches 0. Writing 0 to Timer disables it.

The Cycle and Timer registers can be read using the existing ***rdcycle*** and ***rdtime*** pseudo instructions (***csrrs*** using x0). Also, they can be written using the the **csrrw** instruction (the following pseudo instructions will be added through assembler macros: ***wrcycle*** and ***wrtime***)

# The System

## SoC Builder



## JSON SoC Format



## Software Development

GNU toolchain

Real-Time OS (RTOS)

**To do:**

* Add basic debug support (SPI Interface): Step, BP, Watches, …
* Implement the CPU and a minimum set of components into the following FPGAs
  + XILNIX: ARTIX 7 and ZYNQ
  + Lattice: ICE40 HX8K and HX1K
  + Altera: Cyclone V (TBD)
* Validate/Develop the following IP’s: UART, SPI master, PWM, RTC, AES Accelerator, WDT, …

## I/O IPs

* Analog
  + ADC
  + DAC
  + Comparator
* Digital
  + PWM
  + Port
  + WDT
  + Timer/Counter
  + RTC
* Comm
  + I2C
  + SPI
  + UART
  + USB2??

Interrupt test case

Ecall instruction

Ebreak instruction

Dummy loop and