**3-Stage Pipelined RV32i CPU for Resources-Constrained**

**Embedded 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

## 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)

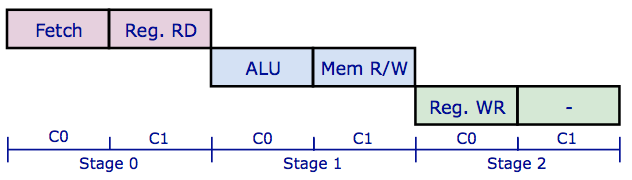


Figure 1. The CPU Pipeline Stages

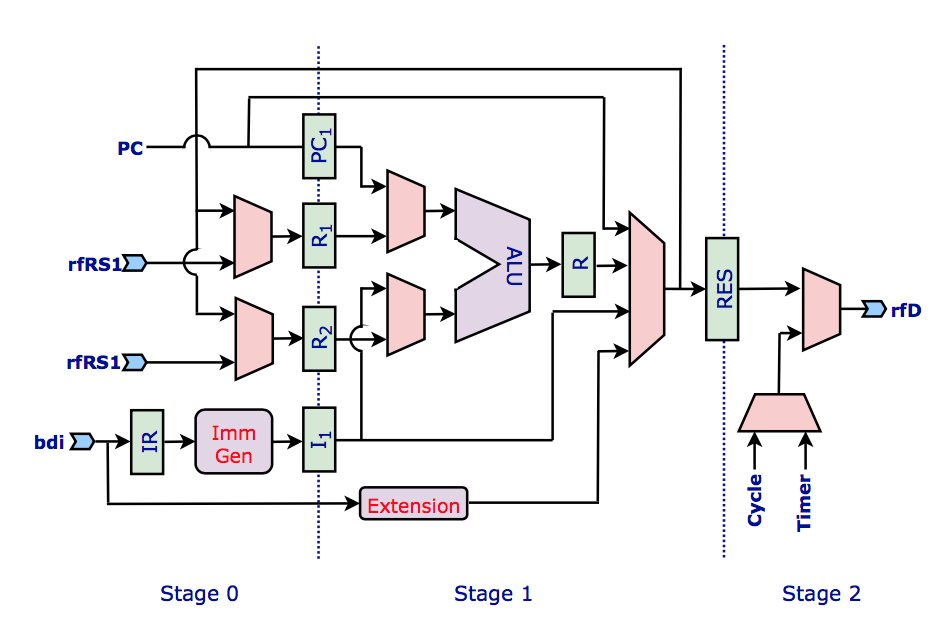


Figure 2. Simplified CPU Datapath showing the Pipeline Stages

## The CPU Interfaces

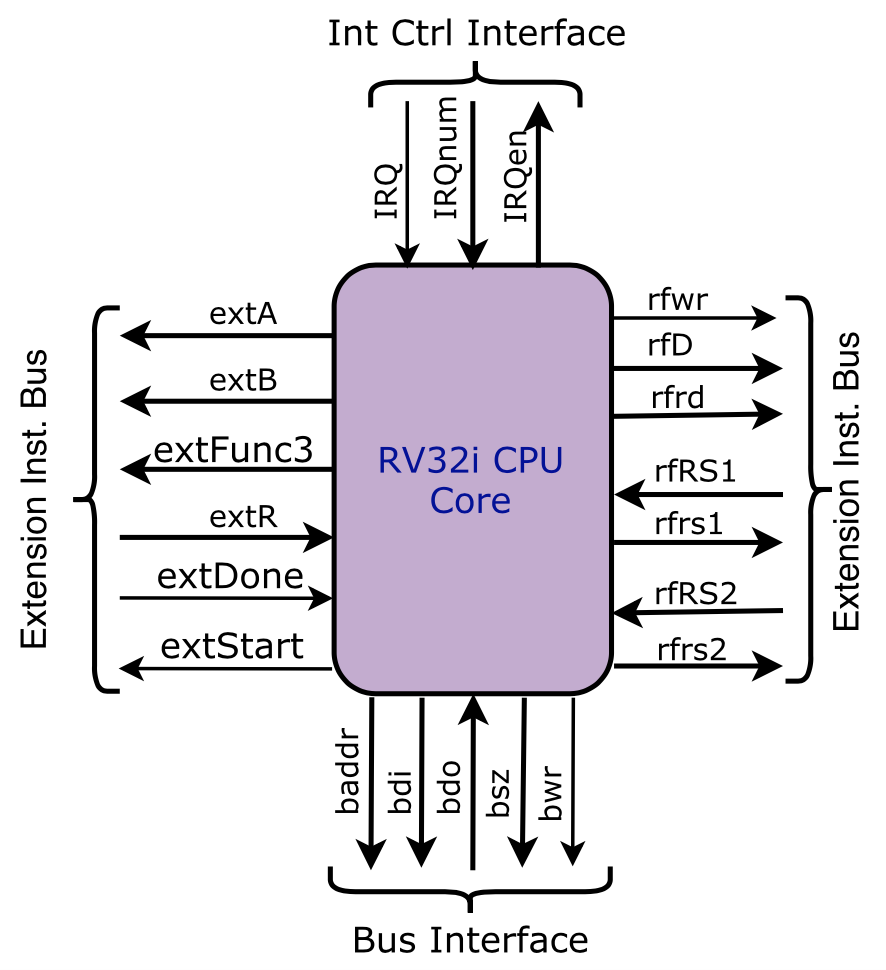


Figure 3.

## 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).

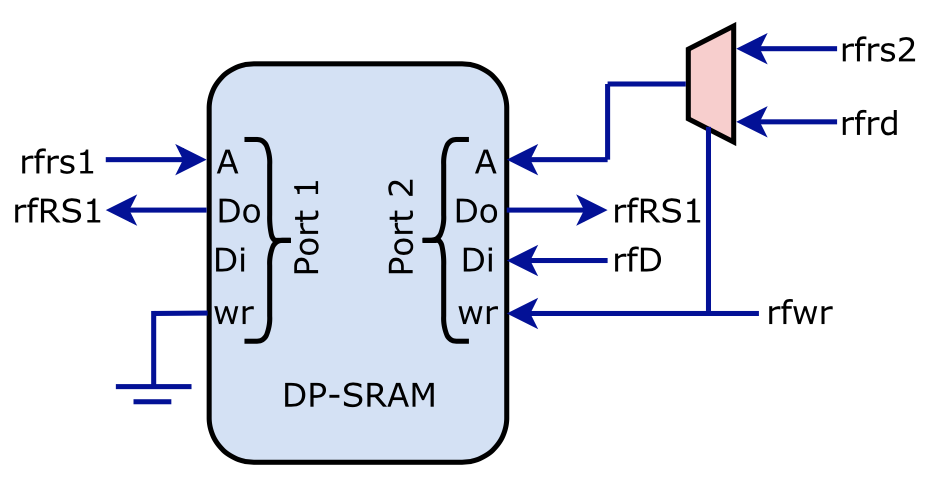


Figure 4. Using DP-SRAM as a Register File

## 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

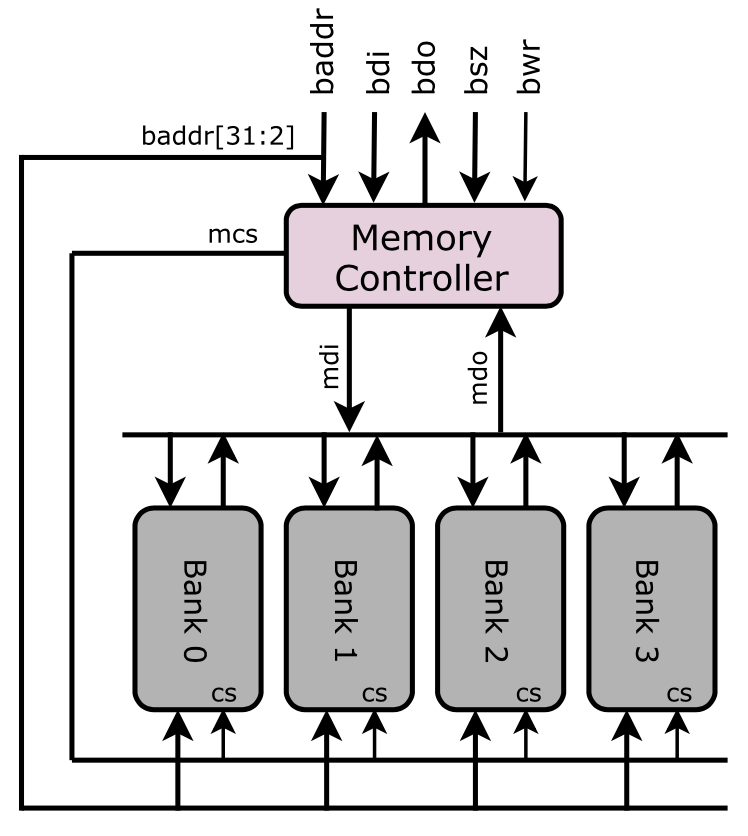


Figure 5. CPU Memory Interface

## Expansion Port

Adding support for new instructions (e.g., multiplication) through non-standard ext. The new instructions have opcode 10\_001. The func3 field is used to specify the instruction. The current implementation provides an output control signal, extStart, which gets set whenever an extension instruction is fetched and decoded (during the second cycle of the first stage of the pipeline). The CPU then *halts* until the extension instruction has been executed (the extension unit should then produce a HIGH signal to the CPU input extDone). As the CPU halts, the instruction following the extension instruction is fetched but does not move down the pipeline until the CPU resumes. Moreover, the data transfer happens through the buses extA, extB, and extR; extA and extB are provided by the CPU and are the data stored in the registers referred to by the rs1 and rs2 fields. When the CPU resumes after an extension instruction, the result should be connected to the extR input bus.

## 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 halts the CPU until an interrupt occurs.
* URET instruction is implemented to return from an ISR

The general behavior of the implemented interrupts system is as follows. When the interrupt flag is raised (set), the CPU switches to a special mode of operation (ISR Mode) at the end of the first cycle (C0) of a pipeline stage, just as a new instruction is fetched. However, similar to the jumps and taken branches, the pipeline is flushed to move the control flow to the appropriate ISR. The address of the instruction to be executed after handling the interrupt is stored in a special-purpose register, ePC, which is used later by the ***uret*** instruction to resume the program execution.

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.

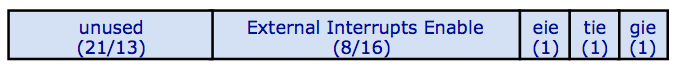


Figure 6. UIE Register Format

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

For external interrupts, however, a multivector is used and contains 16 jump instructions (for 16 possible external sources of interrupts). The mechanism for external interrupts is as follows: an external device requests an interrupt through a signal passed to an interrupt controller that receives interrupt requests from peripherals on one side and translates that into two signals, IRQnum and IRQ. The IRQnum is basically the number/id of the device (from 0 to 15 in this case) and and IRQ signal is passed to the CPU only if that device number is allowed to generate interrupts (controlled through the enable bits of the UIE register). The CPU then sets the interrupt flag and calculates the appropriate effective address of the jump instruction in the multivector mentioned above. The jump instruction leads to a symbol named IRQ**X** where **X** is the id of the source (IRQnum) and should be provided by the user; otherwise, a simple function with only **uret** is provided by default. It is essential, of course, for the provided interrupt handler to end with **uret.**

|  |  |
| --- | --- |
| **Interrupt Source** | **ISR Starting Address (decimal)** |
| ECALL instruction | 16 |
| EBREAK instruction | 32 |
| Timer | 48 |
| External Interrupts: |  |
| IRQ0 | 64 |
| IRQ1 | 68 |
| IRQ2 | 72 |
| ... | ... |
| ... | ... |
| ... | ... |
| IRQ14 | 120 |
| IRQ15 | 124 |

## Counters

Two main 32-bit registers, Cycle and Time registers, are supported. 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.

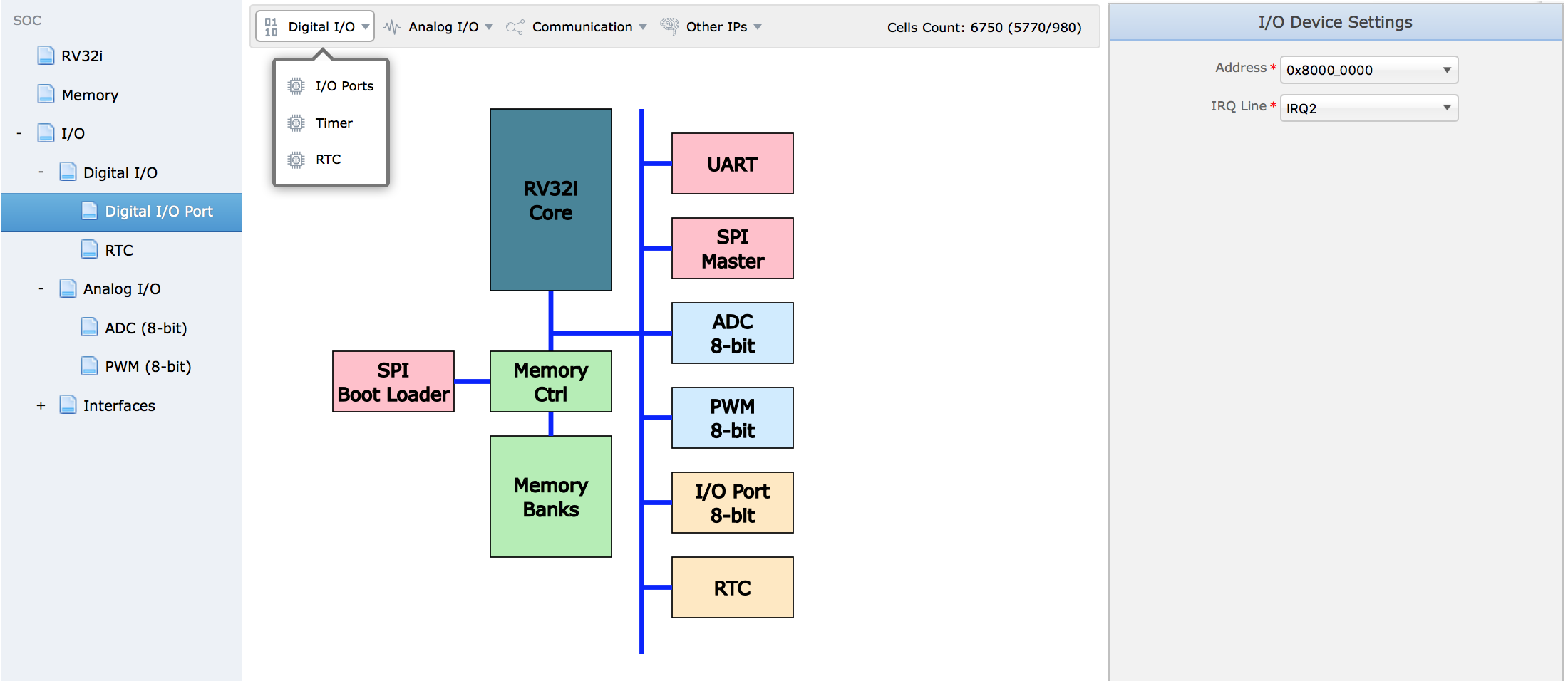
The Cycle and Timer registers can be read using the existing ***rdcycle*** and ***rdtime*** pseudo instructions (***csrrs*** using x0). The read values using these instructions are always two units apart from those at fetch time; more precisely, assuming the initial values in the Timer and Cycle registers, just *after* ***rdcycle/rdtime*** are fetched, were T0 and C0 , respectively, then the values stored in the destination registers would be T0 – 2 (or 0 if T0 < 3) and C0 + 2.

Also, both registers can be written using the the **csrrw** instruction (the following pseudo instructions will be added through assembler macros: ***wrcycle*** and ***wrtime***)

Additionally, a secondary counter of retired instructions is included (Instret) to count non-trivial instructions fully executed by the CPU. It, therefore, gets incremented whenever a non-NOP instruction leaves the last stage of the pipeline. It is usually used for benchmarking and profiling purposes and can be read using the ***rdinstret*** pseudo instruction.

# 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 cases

Ecall instruction

Ebreak instruction

Dummy loop and

## Basic SoC

In this subsection, a basic implementation of a possible SoC supporting external interrupts and I/O will be described.

[diagram here (WIP)]

The components of the SoC:

* The CPU with all of its interfaces (except the extension interface), see Figure 3
* A basic bus:
  + The 5 channels seen at Figure 3:
    - **baddr:** Address of the transaction
    - **bdi:** Input data for a slave (memory, I/O registers, …)
    - **bdo:** Output data for the master (CPU)
    - **bsz:** Data size(00: byte, 01: half-word, 10: word(4 bytes)
    - **bwr:** Write enable
* I/O registers:
  + Each attached to the bus through a tri-state buffer enabled when targeted for a transaction by the master (0x800000x addresses).
  + The output of these registers are exposed to peripherals.
* Memory:
  + Also be attached to the bus through a tri-state buffer enabled when targeted for a transaction(for a memory size of 4KB, 0x00000xxx addresses).
  + [Wait states here?]
* Register file:
  + Connected to the CPU directly through the interface.
* Interrupts Controller:
  + Connected directly to the CPU through its interface.
  + Receives from the CPU (the UIE register) the enable signals.
  + Its other input line (INT) is exposed to peripherals (up to 16).