Table of Contents

[Preface 2](#_Toc403361614)

[Who This Book is For 2](#_Toc403361615)

[Goals 2](#_Toc403361616)

[Motivation 2](#_Toc403361617)

[About the Author 2](#_Toc403361618)

[Programming Model 4](#_Toc403361619)

[General Purpose Register Array 4](#_Toc403361620)

[Program Counter 5](#_Toc403361621)

[Stack Pointer 5](#_Toc403361622)

[Segment Selector Registers 6](#_Toc403361623)

[Modifying 6](#_Toc403361624)

[Initial Values 6](#_Toc403361625)

[Kernel Mode Accessible Registers 7](#_Toc403361626)

[Tick Count Register (SPR 00 or TICK) 7](#_Toc403361627)

[Interrupt Vector Table Base Address (SPR 01 or VBR) 7](#_Toc403361628)

[Memory Management 8](#_Toc403361629)

[The Descriptor Table 8](#_Toc403361630)

[Program Code Addressing 9](#_Toc403361631)

[Data Memory Addressing 10](#_Toc403361632)

[High Memory Area 10](#_Toc403361633)

[Interrupts 11](#_Toc403361634)

[Interrupt Vector Table 11](#_Toc403361635)

[Instruction Set Summary 12](#_Toc403361636)

[ADD - Addition 13](#_Toc403361637)

[Register-Immediate Format: 13](#_Toc403361638)

[Register-Register Format: 13](#_Toc403361639)

[Description: 13](#_Toc403361640)

[BSR – Branch to Subroutine 14](#_Toc403361641)

[Instruction Format: 14](#_Toc403361642)

[Description: 14](#_Toc403361643)

[JSP – Jump Segment Prefix 15](#_Toc403361644)

[Instruction Format: 15](#_Toc403361645)

[Description: 15](#_Toc403361646)

# Preface

## Who This Book is For

This book is for the enthusiast who’s looking for an instruction set architecture (ISA) suitable for embedded projects; an ISA capable of running a modern operating system. It’s advisable that one have a fairly good background in digital electronics and computer systems before attempting a read. Examples are provided in the Verilog language, it would be helpful to have some understanding of HDL languages. This book (FISA32 – Finitron 32 bit ISA) simply presents the ISA without reviewing the architectural choices.

## Goals

The goal is to come up with a 32-bit embedded controller style ISA. The ISA should be relatively simple to implement, yet powerful enough to support modern operating systems. In particular the author wanted to experiment with memory management models (segmentation), floating point and other things present in many systems outside of the core cpu itself. “adding onto” the core processor creates a larger overall design. In order to fit the design in a smaller FPGA a smaller central core is required.

## Motivation

The author’s prior project , Table888 is a 64-bit processor that fits in a largish FPGA. It was designed around having an easy to read instruction set, and an easy port for a compiler. The design was also done considering future improvements in FPGA size. It’s 64-bit size is difficult to fit into an low-cost FPGA when included for a system-on-chip. (SOC). The FISA32 should result in a much smaller footprint, making it an easier fit into smaller FPGA’s.

## About the Author

First a warning: I’m an enthusiastic hobbyist like yourself, with a ton of experience. I’ve spent a lot of time at home doing research and implementing several soft-core processors, almost maniacally. One of the first cores I worked on was a 6502 emulation. I then went on to develop the Butterfly32 core. Later the Raptor64. I have about 20 years professional experience working on banking applications at a variety of language levels including assembler. So I have some real world experience developing complex applications. I also have a diploma in electronics engineering technology. Some of the cores I work on these days are really too complex and too large to do at home on an inexpensive FPGA. I await bigger, better, faster boards yet to come.

# Design Characteristics

The FISA32 can probably be classified as a hybrid RISC-CISC instruction set. It includes several features of a CISC machine such as an explicit stack pointer and stack operations. However all other memory access is performed with load and store instructions just like a regular RISC machine.

There is some redundancy in the instruction set, which tends to be a feature of CISC machines. FISA32 is not a minimalist instruction set. FISA32 includes things like symmetrical branch operations, and multiply and divide operations.

The ISA supports only a 28 bit address space which is a result of the memory management in use. The amount of addressing matches with an embedded controller style of system. The amount of RAM in embedded systems is usually limited. The test system has a 128MB RAM which requires 27 bit addressing, but many embedded systems have much less.

There is a large 32 entry general purpose register array, typical of a RISC machine.

The ISA provides for a segmented memory management architecture by supporting six segment registers. The segment registers have base and limit control of program segments. The limit control provides a rudimentary form of memory protection. Load / store operations beyond the limit will fail.

# Programming Model

## General Purpose Register Array

The ISA supports 32, 32 bit general purpose integer registers.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| # | Mne. | Usage | Saver | Default Seg. |
| 0 | R0 | Always zero |  |  |
| 1 | R1 | first return value | Caller | DS |
| 2 | R2 | second return value | Caller |
| 3 | R3 | subroutine arguments | Caller |
| 4 | R4 | Caller |
| 5 | R5 | Caller |
| 6 | R6 | Caller |
| 7 | R7 | Caller |
| 8 | R8 | Caller |
| 9 | R9 | Caller |
| 10 | R10 | Caller |
| 11 | R11 | Saved registers | Callee |
| 12 | R12 | Callee |
| 13 | R13 | Callee |
| 14 | R14 | Callee |
| 15 | R15 | Callee |
| 16 | R16 | Callee |
| 17 | R17 | Callee |
| 18 | R18 | Callee |
| 19 | R19 | Callee |
| 20 | R20 | Callee |
| 21 | R21 | Callee |
| 22 | R22 | Temporaries | Caller |
| 23 | R23 | Caller |
| 24 | R24 | Caller |
| 25 | R25 | Caller |
| 26 | TR | task register |  | ES |
| 27 | TP | thread pointer |  | FS |
| 28 | GP | global pointer |  | GS |
| 29 | BP | base / frame pointer |  | SS |
| 30 | SP | stack pointer alias |  | SS |
| 31 | PC | Program counter alias |  | CS |

## Program Counter

The program counter is 23 bits in size. The two LSB’s of the counter are always zero. This allows up to 8MB of sequential code. When combined with the code segment base register 27 bit code addressing is possible.

|  |  |
| --- | --- |
| 22 2 | 10 |
| Offset22..2 | 00 |

The program counter normally just increments by four (which is why it is called a counter), however this behaviour may be altered by a JMP, JSR, RTS, BRK, RTI or branch instruction. Refer to the instruction descriptions to see how these instructions modify the program counter.

The program counter is readable as general purpose register #31 for use in forming program counter relative addresses.

On reset the program counter is initialized to $10000.

## Stack Pointer

The ISA features an explicit stack pointer which is 23 bits in size. The two LSB’s of the stack pointer are always zero. This allows a stack space of up to 8MB. When combined with the stack segment addressing of up to 27 bits is possible. The stack pointer increments or decrements by four as values are pushed and popped from the stack. The stack pointer is always word aligned.

|  |  |
| --- | --- |
| 22 2 | 10 |
| Offset22..2 | 00 |

An alias of the stack pointer is available from the general purpose register array as register r30. It may be used for performing load and store operations from the stack.

The stack pointer register (R30) is automatically updated by instructions that use the stack (JSR, RTS, BRK, RTI, PUSH, and POP).

The stack pointer contents are undefined on reset. The stack segment, selector, and pointer should all be set in by the reset routine before any subroutine calls or interrupts are enabled.

Interrupts are globally disabled at reset until the first time the stack pointer is updated.

## Segment Selector Registers

The ISA supports six ten bit segment selector registers as shown in the table below.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| SPR | SEG | 31 10 | 9 0 |  |
| 18 | 0 |  | DS10 | data selector |
| 19 | 1 |  | ES10 |  |
| 1A | 2 |  | FS10 |  |
| 1B | 3 |  | GS10 |  |
| 1C | 4 |  | --- | unsupported |
| 1D | 5 |  | --- | unsupported |
| 1E | 6 |  | SS10 | stack selector |
| 1F | 7 |  | CS10 | code selector |

The segment selector register used for address formation is selected by a bit field encoded in the instruction. The assembler will setup a default selector corresponding to the general purpose register used. For instance if the instruction uses the stack pointer, the assembler will automatically select the stack selector as the segment register for the instruction. This default setup may be overridden using a selector prefix in assembler code. The prefix does not output any additional opcodes, it just sets the segment selector register number in the current instruction.

### Modifying

The selector registers are accessed via the MTSPR and MFSPR instructions. It is not possible to modify the code selector by directly moving a value to it via MTSPR. The code segment selector register must be modified using one of the control flow instructions that may affect it (JMP, JSR, RTS, RTI, or BRK).

### Initial Values

Except for the code segment, the segment registers contain undefined values on reset. On reset the code segment selector register is set to $000.

## Kernel Mode Accessible Registers

### Tick Count Register (SPR 00 or TICK)

This register contains a count of the number of clock cycles that have passed since the last time the processor was reset. Tick may be used for high-resolution timing or performance measurement.

### Interrupt Vector Table Base Address (SPR 01 or VBR)

This register contains the physical base address of the interrupt vector table in memory. The Table is 2kB aligned.

|  |  |
| --- | --- |
| 31 11 | 10 0 |
| Address31..11 | 00011 |

On reset the content of this register is undefined. It must be set by the reset routine before interrupts are enabled.

# Memory Management

The ISA provides for a segmented memory management model. Memory is divided into a number of segments. The segments are described in the segment descriptor table. A 10 bit selector value is used to index into the descriptor table in order to retrieve the information on the segments.

## The Descriptor Table

The descriptor table contains information on the location and size for segments. This table is a special 1k word dual-ported memory. Entries in the table have the following format:

|  |  |  |
| --- | --- | --- |
|  | 31 16 | 15 0 |
| w0 | Limit16 | Base16 |
| w1 | Limit16 | Base16 |
| … |  |  |

The descriptor table is located at a fixed address of $0000\_1000.

10 bit selectors are used to index into the table in order to determine the characteristics of the segment.

## Program Code Addressing

The ISA supports a 27 bit code address space. This was chosen for the FPGA board which has 128MB (27 address bits) of memory onboard. A sixteen bit segment base value is shifted left eleven bits and combined with a twenty-three bit offset value to form a twenty-seven bit address. The segment paragraph size is 2kB. It may be desirable to reduce the paragraph size for systems with a smaller memory footprint.

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | |  |
| 09 | Offset22.2 | | 00 |
|  | + | |  |
| Base15.0 | | 000010..0 | |
|  | = | |  |
| Address26..0 | | | |

The JSR instruction pushes both the code segment selector and the program counter on the stack. Both of these values can be pushed onto the stack in a single word.

|  |  |  |
| --- | --- | --- |
| 31 | 30 21 | 20 0 |
| 0 | Selector9..0 | Offset22..2 |

The RTS instruction always restores both the code segment selector and the offset.

The JMP and JSR instruction transfer control to an address within the current segment. A 23 bit address range is supported by these instructions, allowing code of up to 8MB in size within a single segment. In order to transfer to code in another segment the JSP (jump segment prefix) instruction is issued just prior to a JMP or JSR.

## Data Memory Addressing

The ISA supports a 27 bit data addressing range (this is the amount of memory available on the FPGA board in use). A sixteen bit segment base value is combined with up to 26 bits of displacement from a load or store instruction to form a final 27 bit address.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | | |  |
| 01 | Offset25..0 | | | |
|  | + | | |  |
| Base15.0 | | | 00010..0 | |
|  | | = | |  |
| Address26..0 | | | | |

## High Memory Area

When the offset plus the segment base exceeds 27 bit addressing, the high memory area is active. For instance, by setting the code segment value to a maximum ($FFFF) the processor may reach into the high memory area up to 8MB-2kB. This may be a useful storage place for ROM and IO, leaving a full 128MB for other memory requirements. In fact, on reset the code segment base is set to $FFFF so it naturally points to the last 2kB paragraph of memory.

# Interrupts

An interrupt causes the code selector and program counter to be saved in the return address format on the stack. Next the code segment selector and program counter are loaded from the interrupt vector table. These two items are stored in the vector table in the same format as a return address.

|  |  |  |
| --- | --- | --- |
| 31 | 30 21 | 20 0 |
| 0 | Selector9..0 | Offset22..2 |

The interrupt mask is stored in a backup register, then further interrupts are masked from happening. The processor mode is stored in a backup register then the processor is switched to kernel mode.

Interrupts are globally disabled at reset until the first time the stack pointer is updated.

## Interrupt Vector Table

The interrupt vector table stores the addresses of interrupt routines. The vector table holds 512, 4 byte vectors making the table 2kB in size. The location of the interrupt vector table is specified by the vector table base address register (VBR). The location specified is a physical address not subject to segmentation.

# Instruction Set Summary

## ADD - Addition

ADDI #imm

ADD Rt, Ra, Rb

### Register-Immediate Format:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Immed16 | Rt5 | Ra5 | 046 | ADDI #imm |

Rt = Ra + imm

### Register-Register Format:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 046 | ~5 | Rt5 | Rb5 | Ra5 | 026 | ADD Rt,Ra,Rb |

Rt = Ra + Rb

### Description:

The sum of a register and immediate value, or register and register value are placed into the target register.

## BSR – Branch to Subroutine

BSR label

### Instruction Format:

|  |  |  |  |
| --- | --- | --- | --- |
| Displacement22..2 | 315 | 156h | BSR |

### Description:

This is an alternate mnemonic for the JSR instruction where the register to use in address formation is the program counter. The program counter is added to the displacement to form the target address. The program counter is pointing to the address of the instruction following the BSR. So the relative address is relative to the address of the next instruction.

## JSP – Jump Segment Prefix

### Instruction Format:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 26 |  | 20 11 | 10 6 | 5 0 |  |
| 3C6 | ~5 | Imm9..0 | Ra5 | 026 | JSP |

### Description:

Either the immediate value or the contents of a register are used to determine the selector value for the following jump or jump to subroutine instruction. Either the immediate value or the register contents should be zero. The two values (immediate and register) are OR’d together to form the selector value. The code segment selector is loaded with the value from this instruction when a following JMP or JSR takes place. Both the program counter and the code segment selector are altered at the same time as an indivisible operation. If a JSP instruction is erroneously issued without a following JMP or JSR the instruction will be treated as a NOP operation and the code segment selector will not be altered.

Interrupts are disabled between the JSP prefix and the following JMP / JSR instruction.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | | |  | | | | |  | |  |  |
| ~17 | | | | | | | | | | vectno9 | | | 006 | BRK |
| Address22..2 | | | | | | | | | | | | Ra5 | Opcode6 | JMP / JSR |
| Immed16 | | | | | | | | | Rt5 | | | Ra5 | Opcode6 | RI format |
| Func6 | | | ~5 | | | | Rt5 | | Rb5 | | | Ra5 | 026 | RR format |
| Func6 | | | ~5 | | | | Rt5 | | Imm5 | | | Ra5 | 026 | Shift # |
| 386 | | | ~2 | b3 | | | Rc5 | | Rb5 | | | Ra5 | 026 | PUSH |
| 396 | | | ~5 | | | | Rt5 | | ~5 | | | ~5 | 026 | POP |
| 3C6 | | | ~5 | | | | Imm10 | | | | | Ra5 | 026 | JSP |
| 3F6 | | | ~5 | | | | ~5 | | ~5 | | | ~5 | 026 | RTI |
| Disp13 | | | | | | | | Op3 | Rb5 | | | Ra5 | Opcode6 | Bcc / Bccu |
| Disp13 | | | | | | | | Immed8 | | | | Ra5 | Opcode6 | Bcc # |
| Disp13 | | | | | | | | Sg3 | Rt5 | | | Ra5 | Opcode6 | Lx |
| Disp13 | | | | | | | | Sg3 | Rb5 | | | Ra5 | Opcode6 | Sx |
| Disp6 | | Sg3 | | | Sc2 | | Rt5 | | Rb5 | | | Ra5 | Opcode6 | LxX |
| Disp6 | | Sg3 | | | Sc2 | | Rc5 | | Rb5 | | | Ra5 | Opcode6 | SxX |
| D3 | Sg3 | Rt5 | | | | | Rc5 | | Rb5 | | | Ra5 | 2E6 | CAS |
|  | | | | | | | | |  | | |  |  |  |
| Immediate26 | | | | | | | | | | | | | 256 | PUSHC |
| Immediate26 | | | | | | | | | | | | | 3E6 | IMM |

Major Opcode

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | x0 | x1 | x2 | x3 | x4 | x5 | x6 | x7 | x8 | x9 | xA | xB | xC | xD | xE | xF |
| 0x | BRK |  | {RR} |  | ADD# | SUB# | CMP# | MUL# | DIV# | MOD# |  |  | AND# | OR# | EOR# |  |
| 1x | BEQ# | BNE# | {Bcc} | {Bccu} | JMP | JSR | CMPU# |  | BLT# | BGE# | BLE# | BGT# | BLTU# | BGEU# | BLEU# | BGTU# |
| 2x | LB | LBU | LH | LHU | LW | PUSHC | SEQ# | SNE# | LBX | LBUX | LHX | LHUX | LWX | LEAX | CAS |  |
| 3x | SLT# | SGE# | SLE# | SGT# | SLTU# | SGEU# | SLEU# | SGTU# | SBX | SHX | SWX | SB | SH | SW | IMM |  |

02 Group Func

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | x0 | x1 | x2 | x3 | x4 | x5 | x6 | x7 | x8 | x9 | xA | xB | xC | xD | xE | xF |
| 0x | NAND | NOR | ENOR | ORC | ADD | SUB | CMP | MUL | DIV | MOD |  |  | AND | OR | EOR | ANDC |
| 1x |  |  |  |  | ADDU | SUBU | CMPU | MULU | DIVU | MODU |  |  |  |  | MTSPR | MFSPR |
| 2x | SLL | ROL | SRL | ROR | SRA |  | SEQ | SNE | SLT | SGE | SLE | SGT | SLTU | SGEU | SLEU | SGTU |
| 3x | SLL # | ROL # | SRL # | ROR # | SRA # |  |  |  | PUSH | POP |  |  | JSP |  | RTS | RTI |

12 Group Op3

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | x0 | x1 | x2 | x3 | x4 | x5 | x6 | x7 |
| 0x | BEQ | BNE |  |  | BLT | BGE | BLE | BGT |

13 Group Op3

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | x0 | x1 | x2 | x3 | x4 | x5 | x6 | x7 |
| 0x |  |  |  |  | BLTU | BGEU | BLEU | BGTU |