# COMP22712
# Microcontrollers
# Laboratory Manual
# 2025 RISC-V Edition

Department of Computer Science
University of Manchester

January 18, 2025

# Contents


**1 Introduction** **3**

**2 Organisation of the Laboratory** **5**

**3 The Laboratory Equipment** **7**

**4 Programming Style & Practice** **24**

**5 Exercise 1: Simple Output** **28**

**6 Data-driven and Code-driven Algorithms** **35**

**7 Exercise 2: Less Simple Output** **37**

**8 Programming Hints** **42**

**9 Stacks: "Why?" ... and "How?"** **47**

**10 Exercise 3: Nesting Procedure Calls** **51**

**11 Code Organisation** **55**

**12 RISC-V Operating Modes** **58**

**13 Memory protection** **65**

**14 Exercise 4: System Calls** **68**

**15 Exercise 5: Counters and Timers** **74**

**16 Calling Conventions** **80**

**17 Exercise 6: Interrupts** **82**

**18 Shifts and Rotates** **90**

**19 Exercise 7: Key Debouncing and Keyboard Scanning** **93**

**20 Relocatability** **98**

**21 Exercise 8: System Design and Squeaky Noises** **100**

**22 Exercise 9: Project** **104**

**23 RISC-V Assembly Language Mnemonics and Directives** **106**

**24 References** **110**

**25 Appendix A: Keypad explained** **112**

**26 ASCII Character Set** **113**

# 1 Introduction

## 1.1 Purpose

The purpose of this module is to:

- gain more experience and confidence in assembly language programming
- learn something of the structure of the low-level embedded software
- probe the hardware/software interface
- interface computers to their environment

## 1.2 Prerequisites

The module will require some knowledge of assembly language programming and of basic logic design. It is assumed that participants will already have taken COMP15111 and a module in basic digital design (COMP12111). Some of the tools used in these modules will be revisited in COMP22712 — you should therefore bring your own manuals and notes from these to refer to.

This is primarily a software module, although a significant amount of 'hardware literacy' is involved.

## 1.3 Teaching Style

This manual describes the entire module, together with some extensions and background material; we recommend you read it (thoroughly) although not all at once! There are no formal lectures although there will be interludes of taught material within the labs. However this is a practical subject and the only way to learn it is by having a go (and by making mistakes).

Because of this style all the assessment is also done on the practical work. This means that you must not only complete the module but you must hand in any required evidence so that we know you have done it.

Submit the appropriate exercises by e-mail copied to:

`james.garside@manchester.ac.uk`
`ainur.begalinova@manchester.ac.uk`

The exercises in this module are intended to introduce a set of concepts. In general it is necessary to complete the basic exercise before moving on, as some of the ideas — and frequently some of the practical output — are prerequisites for later exercises. Many exercises include suggestions for extensions; such extensions are not necessary to the module but are there to provide a starting point if you want to learn more.

Like a magazine, this manual has a number of interspersed 'side boxes'. These contain material which is not *directly* relevant in that place but should become in useful about that time. Each attempts to encapsulate a particular concept which should contribute to both this laboratory and other later modules. Primarily they are there to make your progress both easier and more interesting; there are also cases where ignoring their advice will probably waste time (or marks).

You will find occasional 'RV32 programming puzzles' filling up odd spaces in the manual. These are purely fun challenges for you to see how close to the target answer you can get. Some of these are really quite hard; if you can work out an answer without help you can feel suitably smug!

## 1.4 Core Philosophy

The module is focussed on **computer interfacing** with some of the techniques involved: in particular some real-time aspects such as **interrupts**. A selection of **I/O** devices are introduced and a little hardware development is included.

The programming is done in RISC-V **assembly language**; this does not exactly represent the usual contemporary, industrial flow, where the vast majority of the coding is likely to be in C (or C++), but usually helps enormously with the understanding of what is happening inside the machine. Translation of the *principles* to a programming language in future will be easy.

## 1.5 External References

There are a number of external references scattered through this manual. These may be to teaching resources, research projects, product data etc. and are usually to information obtainable via the World Wide Web (WWW). In general URLs are not given — WWW pages sometimes disappear and new and better ones are created — instead you should track down your own sources with the search engines of your choice. There is a very powerful research resource literally at your fingertips; use it!

## 1.6 Revised syllabus

This manual has been produced for the first year of a major revision to this module. The basic exercises are mostly the same as we have run in the past – so we know they are appropriate – with only some details updated. The equipment is new (& modernised) but that difference is fairly minor; the major change is the processor Instruction Set Architecture (ISA) has been modernised from the (now outdated) 32-bit ARM to the contemporary RISC-V [`https://riscv.org/`] a change which the CS department is to make 'across the board'. The original plan was that you year would have studied the RISC-V ISA (rather than ARM) in the first year. We've added some 'conversion' material since this plan didn't get fulfilled. Please be patient whilst we debug this and think of it as a two-for-one bonus!

We appreciate that this will be a new ISA for most people here but it adds and subtracts, loads, stores and ~~branches~~ jumps just like an ARM. You should pick it up quite easily: there is an introduction in section 3.1 and a summary in chapter 23.

## 1.7 Errata

Finally, this is a new edition of this module with a rewritten manual. When you spot something wrong (and there will be despite all efforts), or something not clear, please feed that back to the staff and we'll try to improve it.

Both staff and future students may be grateful to you.

**Acknowledgements**

The Department of Computer Science would like to thank the AMD/Xilinx University Program for their donation of FPGAs for the equipment used in this module: particular thanks for assistance go to Dr. Mario Ruiz and Cathal McCabe from AMD Research and Advanced Development.

# 2 | Organisation of the Laboratory

Organisation will be as informal as possible. All sessions are two hours long and will be performed in the Tootill 0 lab. (LF 9). Most sessions will consist of a short talk on aspects of the particular exercise followed by time spent in practical work. If required/desired there will also be time allocated to a discussion of preceding exercises and a chance for feedback.

Please make every effort to arrive on time. Talks, when they happen, tend to be at the session's scheduled start time.

Some other points:

- Certain exercises require paperwork (listings or schematics) to be handed in. You must do this or you will not be credited for that piece of work.
- Many exercises depend on previous work. You should therefore complete exercises in the order given.
- The syllabus assumes that you will spend approximately the same time working on this module outside laboratory time as you do in scheduled lab. hours. The University suggests that each 'credit' is 10 hours[1] work: 10 credits = 100 hours.
  As there are no lectures to read up on and **no examination** to revise for you should spend this time on the lab. Don't expect to keep up if you don't!
- Some exercises include suggestions for further work. These carry no extra marks but may provide some more good practice. If you have time you may attempt some of these; if not then they are a distraction so don't bother.

## 2.1 Assessment

Marks will be accrued for successfully demonstrating a working design, for efficiency of the solution and for presentation of listings/schematics etc. Not all exercises are formally assessed, but you should still do them!

## 2.2 Organisation of this Manual

There are nine exercises listed here in chapters {5, 7, 10, 14, 15, 17, 19, 21, 22}: these all build *incrementally* on previous work, so complete each before starting the next. We only ask for a subset to be submitted for assessment and feedback but the 'in between' ones are also important.

Some exercises have *suggestions* for extensions, labelled as 'Advanced': only attempt these if you are interested and have time: they are not part of the progression (although the knowledge can still be useful).

In between exercises there are chapters which introduce concepts, many of which are likely to be new to you. We recommend you peruse these chapters as you first work through the manual and, perhaps, return to use them as reference material later.

There are numerous 'side boxes' scattered in what might otherwise be blank spaces. These provide extra concepts, puzzles etc. which can be read or skipped according to taste. We hope some people find some of them worthwhile!

[1] 8 may be more realistic

| Exercise | Sessions | Demonstrate | Submission Deadline (approx.) |
|---|---|---|---|
| 0-Familiarisation | 1 | | |
| 1-Simple Output | 2 | Not assessed | |
| 2-Less Simple Output | 3, 4 | Not assessed | |
| 3-Nesting Procedures | 5, 6 | End of Session 6 | 14/2/2025 |
| 4-System Calls | 7, 8 | Not assessed | |
| 5-Counters/Timers | 9, 10 | End of Session 10 | 28/2/2025 |
| 6-Interrupts | 11, 12 | Not assessed | |
| 7-Keyboards | 13, 14 | End of Session 14 | 14/3/2025 |
| 8-System Design/Sounds | 15, 16 | Not assessed | |
| 9-Project | 17-20 | continued ... | |
| Easter Vacation | | | |
| 9-Project (cont.) | 21-24 | End of Session 24 | 9/5/2025 |

Table 2.1: COMP22712 Laboratory Timetable 2025

### 2.2.1 Introduction to RISC-V

The original plan was to have used this architecture as an example in the first year. As this has not yet happened please **read section 3.1** as soon as possible. We'll give a verbal introduction too, so it's a good idea to turn up on time for your first session.

#### Note on unit prefices

We've endeavoured to use the contemporary (if ugly!) binary prefixes within, where appropriate: thus you should find 'Ki' = $2^{10}$, 'Mi' = $2^{20}$, 'Gi' = $2^{30}$. scattered in here. If any were missed they should be clear from the context. We probably *won't* remember to *say* "kibibytes" etc. every time though!

## 2.3 Equipment Handling Precautions

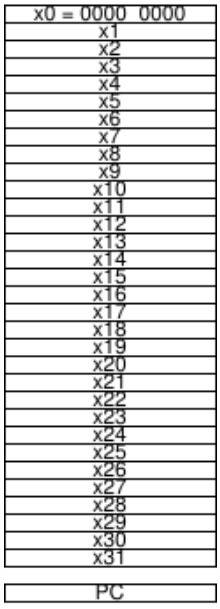
- The computer building is quite susceptible to building up static electric charges on its occupants. Electrostatic Discharge (ESD) will destroy computer chips. Please make every effort to discharge yourself before handling the lab. boards and observe handling precautions as advised by the staff. If you suspect you may be carrying a charge, painless discharge can be achieved by using a coin, key etc. rather than your finger as a conductor to a ground.
- Short circuits may damage certain devices. Please remove any metal watch straps/buckles, rings etc. before handling the boards. There are no dangerous voltages or currents which will harm you here, but this is a good habit to form for when you may be handling other electrical equipment.
- Connecting and disconnecting I/O systems should be done with the power switched off.

# 3 | The Laboratory Equipment

## 3.1 RISC-V Programmers' Model

The RISC-V Instruction Set Architecture (ISA) [`https://riscv.org/`] can be implemented in various levels of detail, starting from a very simple model and being extended in numerous ways. For example, instructions supporting the various floating point standards {32-, 64-, 128-bit} can be included. The RISC-V used here is a fairly simple implementation of an RV32IM – a 32-bit core with only the **M**ultiplication (and division) extension. It includes two *privilege modes* – 'machine' and 'user' so we can sensibly experiment with *exceptions* such as *interrupts*.

### 3.1.1 RISC-V register architecture




The RISC-V has 33 registers with a bit width which depends on the implementation: in the case of the RV32 this is 32 bits. The registers comprise a **P**rogram **C**ounter (`PC`) and a bank of 32 general purpose registers – imaginatively called `x0-x31` – which are all identical except x0 which is 'hard-wired' to `0000_0000`. (All attempts at changing this register will fail.)

However, *by convention* some registers are used for particular purposes and have appropriately aliased alternative names: the most interesting here are:

`x1` Used as a procedure return address ('`ra`')

`x2` Used as the primary stack pointer ('`sp`')

These, and other, register aliases can aid programme organisation; a fuller description of the designers' convention ('ABI') is listed in section 3.3.

The content of `PC` following reset is defined by the implementation: here it is `0000_0000`. All other registers (except `x0`) should be *regarded* as *undefined* at reset time although on a freshly loaded FPGA implementation they will be `0000_0000`.

There are no *flag registers* in RV32I. If a Boolean variable is required it must be stored in an integer register; there are instructions to aid this. However the typical use of flags — for compare/conditional branch as in x86, Arm etc. — is subsumed by a set of compare-and-branch instructions.

### 3.1.2 RISC-V address spaces

In addition to the registers there are two addressable 'memory' spaces: the main memory and the CSR (Control & Status Registers). The CSR space contains various special registers: relevant registers are introduced later, in section 12.3 (see also fig. 12.1). The **main memory** is byte-addressable and has $2^{32}$ byte addresses on the RV32. RISC-V is a *load-store* architecture so memory contents may only be transferred to or from registers when needed for processing. All memory accesses must be 'appropriately' *aligned*: i.e. (32-bit) "words" should only be stored at addresses which are multiples of 4; (16-bit) "halfwords" may be stored at addresses which are multiples of 2; (8-bit) "bytes" can be stored at arbitrary addresses.
If it's of interest here, the system is 'little endian'; if you don't know what that implies, don't worry.

### 3.1.3 RISC-V instruction set

The basic RV32I uses only 32-bit, fixed length op. codes (instructions) which must be word-aligned. (This means the two least-significant bits of `pc` are always '`00`'.)

There are (the usual) three classes of RISC instructions supported:

**Data operations** such as `ADD`, `SUB`, all with two sources and a destination.

**Loads & Stores** moving data from memory to a register, or vice versa.

**Branches & Jumps** RISC-V has three types of branch, chiefly either for making decisions ('branch') or calling and returning from procedures ('jump').

There are a few additional operations but these will be mentioned when relevant. In addition there are quite a lot of *pseudooperations* – convenient ways to express other operations of frequently used short sequences – supported by the tools which can simplify the programmer's task.

**Pseudoinstructions** are statements which a programmer can write, looking like an instruction (syntactically) but are not, directly basic operations. Typically a pseudoinstruction will translate to a single 'real' instruction although, occasionally, they may be short sequences of instructions.

The use of pseudoinstructions can make code simpler to both express and to read. A few of these are *very* useful in coding for RISC-V. Tables below show basic assembler mnemonics: only the **black** items have a directly corresponding instruction; the **grey** mnemonics are translated or augmented with default values.

In the following descriptions:

The **destination** is the first operand (except in 'store' operations).

`xa` etc. represents a processor register: both the '`X*`' and the 'ABI' names can be used.

`###` represents a value as a number or expression (32 bits or (sometimes) fewer).

#### RISC-V data operations

| Operation | Mnemonic | Translation | Remarks |
|---|---|---|---|
| Data ops. | `opr xd, xa, xb` | — | See table 3.2 for operations. |
| Data ops. | `oprI xd, xa, ###` | — | See table 3.2 for operations. |
| NOP | `NOP` | `ADDI zero, zero, 0` | |
| Move | `MV xa, xb` | `ADDI xa, xb, zero` | Register move (copy). |
| Move # | `LI xa, #` | `LUI xa, #####`<br>`ADDI xa, xa, ###` | Load Immediate. |
| Move # (short) | `LI xa, #` | `ADDI xa, zero, ###` | Load Immediate.<br>*This is a non-standard, local variation which will be substituted by the tools.* |
| Load pointer | `LA xa, label` | `AUIPC xa, #####`<br>`ADDI xa, xa, ###` | Load Address (PC relative). |
| Not | `NOT xa, xb` | `XORI xa, xb, -1` | One's complement. |
| Negate | `NEG xa, xb` | `SUB xa, zero, xb` | Two's complement. |
| Comparison | `SLT xa, xb, xc`<br>`SLTU xa, xb, xc`<br>`SLTI xa, xb, #`<br>`SLTIU xa, xb, #` | —<br>—<br>—<br>— | xa := (xb < xc) (signed).<br>xa := (xb < xc) (unsigned).<br>xa := (xb < ###) (signed).<br>xa := (xb < ###) (unsigned). |
| Comparison | `SEQZ xa, xb`<br>`SNEZ xa, xb`<br>`SLTZ xa, xb`<br>`SGTZ xa, xb` | `SLTIU xa, xb, 1`<br>`SLTU xa, zero, xb`<br>`SLT xa, xb, zero`<br>`SLT xa, zero, xb` | True (1) if source is 0.<br>True (1) if source isn't 0.<br>True (1) if source is negative.<br>True (1) if source is positive (non-zero). |

Table 3.1: Data processing mnemonics

Data operations take two operands and produce a result which is written to a register. The destination is the *leftmost* argument. One of the operands is always a register, the other may be a register or a (12-bit, sign-extended) immediate value.

Most of the 'usual' operations are supported: `ADD`, `SUB`, `AND`, `OR` and shifts. The processor used here also has the `MUL`, `DIV` and `REM` extensions since they may, occasionally, be wanted[1].

[1] Note that multiplication and division is significantly more complex than adding and subtracting, so these instructions take multiple execution cycles.

| Operation | Register | Immediate | Remarks |
|---|---|---|---|
| Add | `ADD xd, xa, xb` | `ADDI xd, xa, ###` | |
| Subtract | `SUB xd, xa, xb` | `SUBI xd, xa, ###` | No explicit `SUBI`: uses `ADDI`. |
| And | `AND xd, xa, xb` | `ANDI xd, xa, ###` | |
| Or | `OR xd, xa, xb` | `ORI xd, xa, ###` | |
| Exclusve-Or | `XOR xd, xa, xb` | `XORI xd, xa, ###` | |
| Shift left | `SLL xd, xa, xb` | `SLLI xd, xa, ##` | |
| Shift right | `SRL xd, xa, xb` | `SRLI xd, xa, ##` | **L**ogical shift. |
| Shift right | `SRA xd, xa, xb` | `SRAI xd, xa, ##` | **A**rithmetic shift. |
| Multiply | `MUL xd, xa, xb` | — | Result *mod* $2^{32}$ |
| Multiply | `MULH xd, xa, xb` | — | Result >> 32 (signed) |
| Multiply | `MULHU xd, xa, xb` | — | Result >> 32 (unsigned) |
| Multiply | `MULHSU xd, xa, xb` | — | Result >> 32 (both) |
| Divide | `DIV xd, xa, xb` | — | Result *mod* $2^{32}$ (signed) |
| Divide | `DIVU xd, xa, xb` | — | Result *mod* $2^{32}$ (unsigned) |
| Remainder | `REM xd, xa, xb` | — | Result *mod* $2^{32}$ (signed) |
| Remainder | `REMU xd, xa, xb` | — | Result *mod* $2^{32}$ (unsigned) |

Table 3.2: Data processing mnemonic selection

**RISC-V loads & stores**

| Operation | Mnemonic | Translation | Remarks |
|---|---|---|---|
| Load Word | `LW xa, ###[xb]` | — | 32-bit word; **b**yte & **h**alfword below. |
| Load Halfword | `LH xa, ###[xb]` | — | 16-bit halfword – sign extended. |
| Load Byte | `LB xa, ###[xb]` | — | 8-bit byte – sign extended. |
| Load Half. Uns. | `LHU xa, ###[xb]` | — | As {`LH`, `LB`} above, but |
| Load Byte Uns. | `LBU xa, ###[xb]` | — | loaded value zero extended. |
| Load Word ... | `LW xa, <label>` | `AUIPC xa, #####`<br>`LW xa, ###[xa]` | Also '`LB`', '`LH`', '`LBU`' and '`LHU`'.<br>Extended as above. |
| Store Word ... | `SW xa, ###[xb]` | — | Also '`SB`'and '`SH`'. |
| Store Word ... | `SW xa, <label>, xb` | `AUIPC xb, #####`<br>`SW xa, ###[xb]` | Also '`SB`'and '`SH`'.<br>Second register (`xb`) is 'sacrificial'. |
| Load Address | `LA xa, <label>` | `AUIPC xa, #####`<br>`ADDI xa, xa, ###` | Get *address*: not actual memory load. |
| Load Local Address | `LLA xa, <label>` | `AUIPC xa, #####`<br>`ADDI xa, xa, ###` | Difference seems ... missing! |
| Load Immediate | `LI xa, <value>` | `LUI xa, #####`<br>`ADDI xa, xa, ###` | Value extracted from instruction stream. |
| Load Imm. | `LI xa, <value>` | `ADDI xa, zero, ###` | Short form (*local variant?*) |

Table 3.3: Memory transfer mnemonics

Loads move data from memory to a register. The addressing mode is always specified by a (base) register with a 12-bit, *signed* offset. The mnemonic specifies the size of the quantity moved {**w**ord, **h**alfword or **b**yte}. If the loaded quantity is smaller than the register it will be placed at the least significant end and will be either sign- or zero-extended to the remaining bits as the programmer chooses.

Stores work in a complementary way storing the {**w**ord, least significant **h**alfword or least significant **b**yte} of the chosen register. Only the addressed quantity is changed in memory. The *stored* register is the leftmost argument (which is somewhat inconsistent but typical in assembly language).

In both cases the address must be properly *aligned*: see p. 54 if this is an unfamiliar term.

**Note on {`LW`, `SW`} 'absolute' addressing pseudoinstructions**

E.g. `sw s0, label, t0` Two points (see table 3.3): Firstly, the load 'cheats' but the store needs an independent register to hold the (temp.) address. Secondly, the addresses used are calculated relative to the current `PC` – not 'absolute' addresses.

### RISC-V branches & jumps

RISC-V has two 'classes' of flow control instructions: 'branches' and 'jumps' (table 3.4). These are rather arbitrary nomenclatures. 'Branches' are conditional and take two register operands as well as a destination offset. The registers' contents are compared and the branch is taken only if the condition is met. For example:

```
BGE        x3, x4, target
```

will branch to `target` if `x3` $\geq$ `x4`, else it will continue to the subsequent instruction.

Note that the 'default' assumption is that the values are (two's complement) signed operands: for *unsigned* operands use (e.g.) `BGEU` etc.

The selection of branches in the ISA looks quite limited: for example there is `BLT` ('**B**ranch **L**ess **T**han') but no `BGT`. However `BGT` can still be written and the assembler will swap the operands for you. The assembler will also help with (for example) testing the *sign* of a register by translating to comparisons with `x0`.

'Jumps' are always **J**ump **A**nd **L**ink (`JAL`) which save the address following the instruction in a 'link register'; this enables a simple return to the calling point for procedures. The link register is specified in the op. code; it can be omitted in the assembly language in which case `x1`/`ra` is the default substitute. A simple '`J`' mnemonic will use `x0` which discards the value. The destination offset is also specified in the instruction.

| Operation | Mnemonic | Translation | Remarks |
|---|---|---|---|
| Jump | `J target` | `JAL zero, target` | Discards any 'link' (into `x0`). |
| Jump and Link | `JAL xa, target` | — | Allows any register (`xa`) for PC save. |
| JAL (short) | `JAL target` | `JAL ra, target` | Defaults to 'standard' link register (`x1`). |
| Call (far) | `CALL target` | `AUIPC ra, #####`<br>`JALR ra, ###[ra]` | Subroutine call, assuming `ra` (`x1`) as link. Note: two instructions. |
| Call (short) | `CALL target` | `JAL ra, target` | Subroutine call to 'nearby' routine, assuming `ra` (`x1`) as link. *This is a non-standard, local variation which will be substituted by the tools.* |
| Return | `RET` | `JALR zero, [ra]` | Subroutine return, assuming `x1` as link. |
| Jump Register | `JR xa` | `JALR zero, [xa]` | `PC := xa`; link discarded (into `x0`). |
| Call register | `JALR xa, offset[xb]` | — | `PC := offset + xb`; link in `xa`. |
| Call reg (short) | `JALR xa` | `JALR ra, [xa]` | `PC := xa`; link in `ra`. |
| Branch = | `BEQ xa, xb, target` | — | Branch if `xa` = `xb`. |
| Branch $\neq$ | `BNE xa, xb, target` | — | Branch if `xa` $\neq$ `xb`. |
| Branch < | `BLT xa, xb, target` | — | Branch if `xa` < `xb` (signed). |
| Branch < | `BLTU xa, xb, target` | — | Branch if `xa` < `xb` (unsigned). |
| Branch $\leq$ | `BLE xa, xb, target` | `BGE xb, xa, target` | Branch if `xa` $\leq$ `xb` (signed). |
| Branch $\leq$ | `BLEU xa, xb, target` | `BGEU xb, xa, target` | Branch if `xa` $\leq$ `xb` (unsigned). |
| Branch > | `BGT xa, xb, target` | `BLT xb, xa, target` | Branch if `xa` > `xb` (signed). |
| Branch > | `BGTU xa, xb, target` | `BLTU xb, xa, target` | Branch if `xa` > `xb` (unsigned). |
| Branch $\geq$ | `BGE xa, xb, target` | — | Branch if `xa` $\geq$ `xb` (signed). |
| Branch $\geq$ | `BGEU xa, xb, target` | — | Branch if `xa` $\geq$ `xb` (unsigned). |
| Branch = 0 | `BEQZ xa, target` | `BEQ xa, zero, target` | Branch if specified register (`xa`) is zero. |
| Branch $\neq$ 0 | `BNEZ xa, target` | `BNE xa, zero, target` | Branch if specified register (`xa`) is non-zero. |
| Branch < 0 | `BLTZ xa, target` | `BLT xa, zero, target` | Branch if specified register (`xa`) is negative. |
| Branch $\leq$ 0 | `BLEZ xa, target` | `BGE zero, xa, target` | Branch if specified register (`xa`) is negative or zero. |
| Branch > 0 | `BGTZ xa, target` | `BLT zero, xa, target` | Branch if specified register (`xa`) is positive (inc. zero). |
| Branch $\geq$ 0 | `BGEZ xa, target` | `BGE xa, zero, target` | Branch if specified register (`xa`) is positive (inc. zero). |

Table 3.4: Flow control mnemonics

### Offset addressing

RISC processors typically have fixed-length instruction codes which are no larger (and may be smaller) than their addresses. This means that a full address cannot be contained in a single instruction. Instead, the need to use a register as a *pointer* to the addresses used for loading and storing.

It is the nature of software that data addresses in use tend to be clustered together by normal software organisation.

| Variables | — | Typical RISC-V base register |
|---|---|---|
| Global | Variables accessible across all scopes | GP |
| Local | Stack | SP |
| Arrays | Specified by 'base' address | Various |
| Structures/ Objects | Defined by 'pointer' | Various |
| I/O 'peripheral' | Fixed position | Designated pointer |

In all these cases it is likely that one pointer will point 'nearby' a number of useful items; processor designers assume this is likely. Thus there is typically the ability to **offset** an address from a register so that one (register) pointer (such as a stack pointer) can serve many instructions.

RISC-V is not an exception: there is a 12-bit (signed) offset in the addressing mode for all loads and stores, which gives direct access to addresses within 2K either side of the pointer.

`JALR` moves a specified register into `PC` instead of adding a `PC` offset. The primary use of this is to return from a procedure by recovering the 'link' saved by a preceding `JAL`: thus "`JALR X1`" is the commonest use. This is so common that an alias – `RET` (for '**ret**urn') – is available as an alternative.

The most useful (pseudo)operations to start with are likely to be:

| | |
|---|---|
| Arithmetic/logical: | `ADD(I)`, `SUB(I)`, `AND(I)`, `OR(I)`, `XOR(I)` |
| Register moves: | `MV`, `LI`, `LA` |
| Loads/stores: | `LW` & `SW`, `LB(U)` & `SB` |
| Jumps: | `J`, `JR`, `CALL` & `RET` |
| Conditional branches: | `BEQ` & `BNE`, `BEQZ` & `BNEZ`, `BLT`, `BLE`, `BGE`, `BGT` |

Bear in mind that the range of immediate operands is limited to a 12-bit, sign-extended value (-2048 ·· +2047) or, in most *practical* cases, an 11-bit number (0x000 ·· 0x7FF) for bitwise operations to avoid sign extension difficulties.

#### CSR operations

> This subsection is included here for *reference* purposes. It starts to become relevant from section 12.3. If (as is likely) this is not already familiar it is suggested that it is skipped at first reading.

The most useful CSR mnemonics are probably the simple read (`CSRR`) and write (`CSRW`) operations (both pseudoinstructions). `CSRS` and `CSRC` have some use for CSRs where there is bit encoding (such as with status registers). Beware of the immediate operands since they can only realistically operate on bits 10:0 (12-bit value but sign extended). `CSRRW` (swap) can find some limited use e.g. within a 'trap' service call where it can be used to switch to/from the appropriate privilege level Stack Pointer on entry and exit.

```
CSRRW      sp, MSCRATCH, sp
```

`MSCRATCH` being a 'Machine-level', uncommitted CSR which is rather convenient for this purpose.

| Operation | Mnemonic | Translation | Remarks |
|---|---|---|---|
| Read/write | `CSRRW xa, csr, xb` | — | Read & Write CSR |
| Set | `CSRRS xa, csr, xb` | — | Set CSR bits (& read). |
| Clear | `CSRRC xa, csr, xb` | — | Clear CSR bits (& read). |
| Read/write | `CSRRWI xa, csr, ###` | — | Read & Write CSR immediate. |
| Set | `CSRRSI xa, csr, ###` | — | Set CSR bits immediate. |
| Clear | `CSRRCI xa, csr, ###` | — | Clear CSR bits immediate. |
| Read | `CSRR xa, csr` | `CSRRS xa, csr, zero` | Just read. |
| Write | `CSRW csr, xa` | `CSRRW zero, csr, xa` | Just write. |
| Set | `CSRS csr, xa` | `CSRRS zero, csr, xa` | Just set. |
| Clear | `CSRC csr, xa` | `CSRRC zero, csr, xa` | Just clear. |
| Write | `CSRWI csr, ###` | `CSRRWI zero, csr, ###` | Just write immediate. |
| Set | `CSRSI csr, ###` | `CSRRSI zero, csr, ###` | Just set immediate. |
| Clear | `CSRCI csr, ###` | `CSRRCI zero, csr, ###` | Just clear immediate. |

Table 3.5: CSR mnemonics

**Miscellanea**

The only other instruction in need of highlighting here is `ECALL`. This causes an *exception* which is, in effect, an operating system entry. This will be discussed further when it becomes relevant (chapter 12).
You may have come across this instruction as a 'magic word' in processor simulations: in this module we use the actual instruction function.

There are clear pictures of the full[2] instruction set, complete with extensions on the `https://riscv.org/` website[3]. This is still developing.

## 3.2 Code examples

For 2025 only! Since this is probably the first time you've met the RISC-V instruction set, here are a couple of code examples as illustrations. There are more examples throughout this manual.

```
add_up      la    t0, struct          ; Point at data         (Load Address)
            lw    t1, [t0]            ; Load operand          (Load Word)
            lw    t2, 4[t0]           ; Load operand (offset) (Load Word)
            add   t1, t1, t2          ; Operate               (ADD)
            sw    t1, 8[t0]           ; Store result          (Store Word)
            ret                       ; Return                (Return from CALL)
struct      defw  2, 3, 0             ; Data/result space     (DEFine Word(s))



is_printable lbu   a0, [s0]           ; Load (byte) operand   (Load Byte Unsigned)
             li    t1, 0x20           ; Load constant         (Load Immediate)
             bltu  a0, ta, unprintable ; Conditional skip     (Branch Less Than Unsigned)
             call print_char          ; Subroutine call       (CALL pseudoinstruction)
unprintable  ...
```

[2] As far as it has been designed and ratified.
[3] See, particularly, ⟹ `Technical` ⟹ `Specifications`.

## 3.3 RV32I Application Binary Interface (ABI)

There is a **software convention**[4] for the use of RISC-V registers. For this purpose registers have aliased names. The assembler will understand these names. Other than `x0`/`zero` (and `PC`) the register assignments are quite arbitrary.

You do not have to observe this convention. However it is used implicitly in some assembly language pseudoinstructions and may provide a basis for a 'calling convention'.

**x1/ra** is used to store PC when making procedure calls, assumed by the pseudoinstructions `CALL` and `RET`.

**x2/sp** is conventionally the stack pointer.

**x3/gp** points to global variables. Offsets within the instructions can index individual variables from there.

**x4/tp** pointer for thread-local storage (pthreads/C++).

**x5-x7, x28-x31/t0-t6** are 'scratch' registers; **x5** is sometimes used as a secondary return address at the bottom of the 'call tree'.

**x8-x9, x18-x27/s0-s11** are 'local variables' not changed by procedure calls.

**x10-x17/a0-a7** are procedure arguments; **x10 & x11** (**a0 & a1**) also act as procedure return values.

| ABI name | Register | Use by convention | Preserved? |
|---|---|---|---|
| zero | x0 | hardwired to 0, ignores writes | n/a |
| ra | x1 | **r**eturn **a**ddress for jumps | no |
| sp | x2 | **s**tack **p**ointer | yes |
| gp | x3 | **g**lobal **p**ointer | n/a |
| tp | x4 | **t**hread **p**ointer | n/a |
| s0/fp | x8 | saved register 0 or **f**rame **p**ointer | yes |
| s1 | x9 | saved register 1 | yes |
| s2 | x18 | saved register 2 | yes |
| s3 | x19 | saved register 3 | yes |
| s4 | x20 | saved register 4 | yes |
| s5 | x21 | saved register 5 | yes |
| s6 | x22 | saved register 6 | yes |
| s7 | x23 | saved register 7 | yes |
| s8 | x24 | saved register 8 | yes |
| s9 | x25 | saved register 9 | yes |
| s10 | x26 | saved register 10 | yes |
| s11 | x27 | saved register 11 | yes |
| a0 | x10 | return value or function argument 0 | no |
| a1 | x11 | return value or function argument 1 | no |
| a2 | x12 | function argument 2 | no |
| a3 | x13 | function argument 3 | no |
| a4 | x14 | function argument 4 | no |
| a5 | x15 | function argument 5 | no |
| a6 | x16 | function argument 6 | no |
| a7 | x17 | function argument 7 | no |
| t0 | x5 | temporary register 0 | no |
| t1 | x6 | temporary register 1 | no |
| t2 | x7 | temporary register 2 | no |
| t3 | x28 | temporary register 3 | no |
| t4 | x29 | temporary register 4 | no |
| t5 | x30 | temporary register 5 | no |
| t6 | x31 | temporary register 6 | no |
| pc | (none) | program counter | n/a |

[4] Probably more than one, in the future!

The convention is aimed at supporting compiled code; all compilers (or assembly code writer) obeying the convention will produce interoperable code. The registers are divided into sets, starting with some 'special purpose' registers for language support (e.g. `sp`). 'Temporary' registers are intended for short-term values whilst 'arguments' are for passing values to (and from) `call`ed routines; neither of these sets is guaranteed to return from a `call` unaltered so, if any contents might be wanted in future, they must be "caller saved". 'Saved' registers (in the ABI) are for longer-term local variables and are preserved across `call`s; thus if a(n ABI-compatible) routine writes to them their contents must first be saved and restored before `ret`urning.

The ABI division of register functions may look rather odd. The reason for partitioning the registers this way is (presumably) a result of the inclusion of the '**E**mbedded' (RV32E) 'extension' to the RISC-V which omits x16-x31 (table 3.6) to save core area and power.

| Register | ABI | Use by convention | Preserved? |
|---|---|---|---|
| x0 | zero | hardwired to 0, ignores writes | n/a |
| x1 | ra | return address for jumps | no |
| x2 | sp | stack pointer | yes |
| x3 | gp | global pointer | n/a |
| x4 | tp | thread pointer | n/a |
| x5-7 | t0-2 | temporary register 0-2 | no |
| x8-9 | s0-1 | saved register 0-1 | yes |
| x10-15 | a0-5 | argument 0-5 | no |
| pc | (none) | program counter | n/a |

Table 3.6: RV32E reduced register file

However it is suggested that a different ABI may be produced for this variant, so this is guesswork rather than definitive information.

**RISC-V 'C' extension**

A further consideration may be the '**C**ompressed' extension to the instruction set. This offers alternative, 16-bit instruction encodings for a selected subset of instructions. To save space, *some* 'C' encodings have only 3-bit register specifiers, which are decoded as `x8-x15` (inclusive) i.e. {`s0-s1`, `a0-a5`}, with other, dedicated, operations explicitly using `SP` etc.

If you're interested the details are in the RISC-V instruction set manual (at `riscv.org`).

## 3.4 Software Development Environment

The software development environment used is `bennett`, which is being developed locally. It is a replacement for the system used in COMP15111; the only real difference is that programmes will be downloaded to, and be executed on, a real RISC-V processor rather than a software simulation. This gives access to the real peripheral devices on a stand-alone computer.

It is possible to use other tools to produce object code files if you prefer (there may be minor syntactic differences) although this will restrict the range of debugging displays available. Bennett can load some other file formats but we're not, currently, *actively* supporting other flows.

### 3.4.1 Debugging environment

Bennett provides a monitor front-end and download tool. This allows the state of the remote circuit board to be displayed and manipulated, files to be downloaded into memory and programmes to be executed. The memory and processor register views are updated periodically during execution to aid understanding and debugging. Facilities are introduced gradually in other parts of this manual.

## 3.5 Circuit Board

The microcontroller laboratory is intended to reflect a contemporary development environment used for embedded controllers; in this case everything is loaded onto an FPGA. The basic system comprises a software programmable processor — in this case an RV32IM — with a number of peripheral I/O systems, including parallel I/O, a serial line, a timer and a simple interrupt controller. More I/O is facilitated by exploiting the capacity of the FPGA to suit a particular application; this makes both the software and the hardware programmable and available to the user. The ability to use FPGAs for customisable hardware had considerable influence on the design of systems since the 1990s. As FPGAs have increased in capacity it is now feasible to use them for significant sized systems. Although the comparison is somewhat inexact, the device used in this laboratory (Xilinx Spartan 7 XC7S50) has a capacity of about $2\frac{1}{2}$ million gates.

The PCB (figure 3.1) has a few I/O devices and number of expansion connectors around the board. Two connections are needed for basic operation: a **5 V DC** supply must be connected to the power inlet and a USB connection supplies host communication.

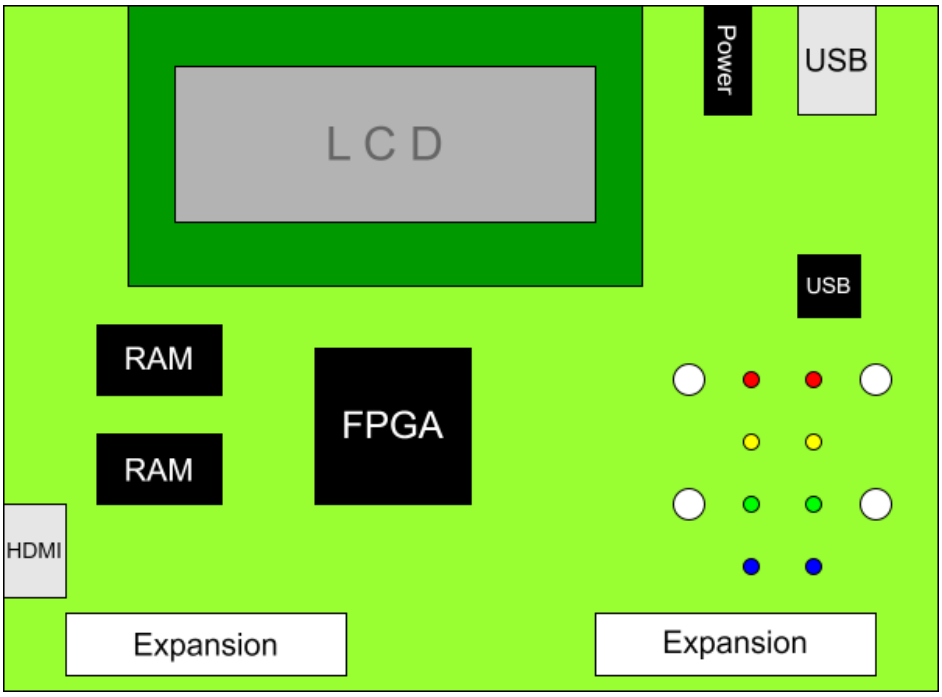


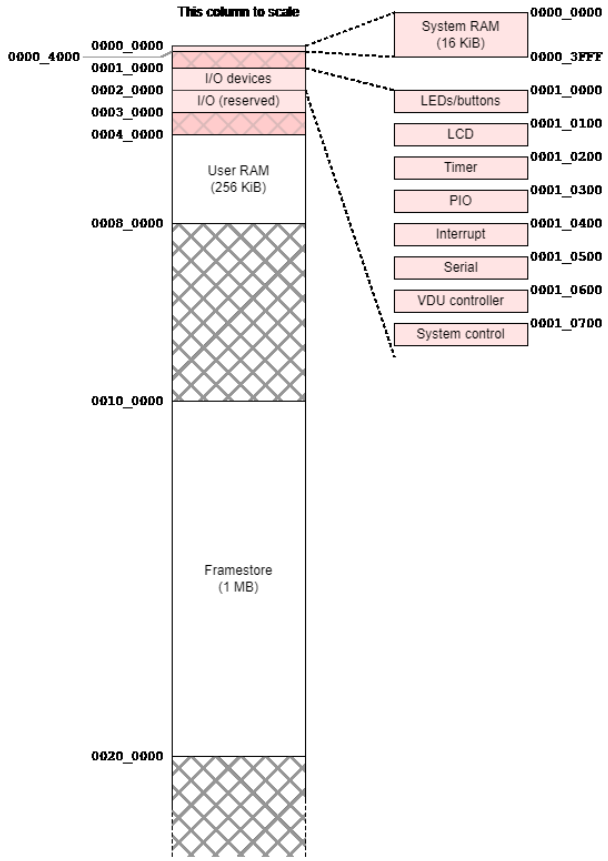

Figure 3.1: The Laboratory PCB connections

The buttons, LEDs and LCD are introduced as they are required for exercises.

The connectors at the bottom allow peripheral expansion which will also feature in a later exercise. These are identical; each has sixteen I/O lines and two each of power (3.3 V) and ground outlets (fig. 3.2).

The initial FPGA configuration is more fully introduced in chapter 21.

### 3.5.1 Lab. board memory map




The memory map is divided into several areas. There are three distinct areas of RAM: two of these areas are implemented on-board the FPGA and run at the same speed as the processor. These are at fixed addresses and are **Tightly Coupled Memories** (TCMs). There is no cache in this system. TCMs are commonly used instead in embedded controllers and deliver the same performance. This system has no SDRAM (or similar) so the TCMs provide 'all' your working memory. The division into two areas is not[5] into instruction and data space – the FPGA 'Block RAM' (BRAM) allows a degree of dual-porting so the RAM can be used for both purposes without penalty – but into a **privileged system space** for 'operating system' code and a **user space** for applications code.

This difference will not matter initially but *will* become important from exercise 4 and beyond.

The third space is a **RAM expansion area of 1 MiB**: this is slightly slower to access. It is used as a video framestore (not explicitly in this module) but can be used for other purposes. It is not privileged (i.e. *is* accessible to applications) *but is only attached to the RISC-V data bus, so you can't run code in that space.*

The address space also contains the peripheral devices ("**memory-mapped I/O**"). These are used for hardware interfacing and require **privileged** (i.e. 'OS') access. The I/O space is subdivided into two areas: one for the devices we have provided and another for your development when we reach exercise 8; this does not affect accessibility.

Each peripheral device has been given a 256 byte (64 word) address space (rather more than needed!) which makes the addresses clearly distinct on a hexadecimal display whilst keeping all I/O registers within $2^{11}$ locations (2 KiB) which is the (positive) immediate offset range of the RISC-V; thus, if you choose, a single I/O pointer register will serve.

Remember that the RISC-V is a 32-bit processor with byte addressing. This means that the addresses of adjacent words are different by 4. Transfers must be *aligned* with the memory so word addresses must be a multiple of 4 (i.e. the lowest two bits of the address must be zero) and halfwords a multiple of 2 (the lowest bit of the address must be zero) — see inset on p. 54. Failure to comply will result in an *alignment* trap.

The 'lowest' $\frac{1}{4}$ MiB of the address space (addresses `0000_0000-0003_FFFF`) is protected and only allows privileged access. This covers 16 KiB of RAM and the memory-mapped I/O space. Unprivileged access to this area will result in an *access fault* trap. Note than the processor starts in this area (at address `0000_0000`) following a reset. The main, $\frac{1}{4}$ MiB RAM area is freely available for general use.

[5] as you *might* have guessed?

**Address aliasing**

You may notice that some ports are visible at more than one address. This is because some address bits are ignored during decoding (thus their state is 'don't care') and the repeat 'period' can be observed to be $2^N$ addresses. This makes the hardware slightly smaller/simpler/faster and is quite common in many systems.

### 3.5.2 I/O port map

The internal I/O region map (**base address** `0001_0000`) is given in table 3.7.

| Offset | Peripheral | Port | Bits | Direction | Size |
|---|---|---|---|---|---|
| 000 | LED interface | LED data (1 = on) | 8 | W | B, H, W |
| 001 | | Switch inputs | 4 | R | B |
| 100 | Character LCD interface | LCD data | 12 | R/W | B, H, W |
| 104 | | Unused | — | — | — |
| 108 | | LCD clear bit(s) | 12 | R/W | B, H, W |
| 10C | | LCD set bit(s) | 12 | R/W | B, H, W |
| 200 | Timer | Counter | 32 | R/W | W |
| 204 | | Limit/modulus | 32 | R/W | W |
| 208 | | Unused | — | — | — |
| 20C | | Control/status | 32 | R/W | W |
| 210 | | Control clear | 32 | W | W |
| 214 | | Control set | 32 | W | W |
| 300 | Parallel Input/Output | PIO data | 32 | R/W | W |
| 304 | | PIO direction | 32 | R/W | W |
| 308 | | PIO clear bit(s) | 32 | R/W | W |
| 30C | | PIO set bit(s) | 32 | R/W | W |
| 400 | Interrupt controller | Interrupt sources | 32 | R | W |
| 404 | | Interrupt enables | 32 | R/W | W |
| 408 | | Interrupt requests | 32 | R | W |
| 40C | | Interrupt mode | 32 | R/W | W |
| 410 | | Interrupt edge | 32 | R | W |
| 410 | | Interrupt edge clear | 32 | W | W |
| 414 | | Interrupt edge set | 32 | W | W |
| 418 | | — | — | — | — |
| 41C | | Interrupt output | 1 | R | W |
| 500 | Serial interface | Data register | 8 | R/W | B, H, W |
| 504 | | Control/status | 8 | R/W | B, H, W |
| 600 | Video controller | ????????? | +++ | ???? | ? |
| 604 | | Under development | +++ | ???? | ? |
| 608 | | Not used here | +++ | ???? | ? |
| 60C | | ????????? | +++ | ???? | ? |
| 700 | Version | Date: 24-bit BCD | 32 | R | B, H, W |
| 700 | Halt | Halt port | 32 | W | B, H, W |
| 704 | Clock speed | Clock speed | 32 | R | W |
| 708 | Pin function | Signal source/destination | 32 | R/W | W |
| 70C | — | — | — | — | — |
| 710 | System time | Timer (L) | 32 | R/W | W |
| 714 | | Timer (H) | 32 | R/W | W |
| 718 | | Timer cmp (L) | 32 | R/W | W |
| 71C | | Timer cmp (H) | 32 | R/W | W |

Table 3.7: Lab. board *provided* peripheral offsets

**LED port: base address `0001_0000`**

This is a simple, fixed-function parallel I/O port. It comprises a single, 12-bit data port: the lower 8 bits correspond to the 8 LEDs and are always output bits whilst bits 11:8 are input bits from the four push-buttons. The LED states can be read back but *writing* to the buttons will have no effect.

This port **allows byte access** so the buttons (or LEDs) can be read alone with an `LB` (or `LBU`) instruction at the appropriate address.

**LCD port: base address `0001_0100`**

The LCD port is a 12-bit interface as shown in figure 3.8. These signals correspond to the pins on the HD44780 alphanumeric LCD module. All these pins are outputs except the data bus, which is bidirectional so the module's registers can also be read. The data direction is controlled internally (automatically) by the state of the `R/W̄` bit.

| Bit(s) | Use | States |
|---|---|---|
| 11 | Backlight enable | 1 = On |
| 10 | Enable (`E`) | 1 = Enabled |
| 9 | Register Select (`RS`) | 1 = Display |
| 8 | Read, not Write (`R/W̄`) | 1 = Read |
| 7-0 | Data bus | — |

Table 3.8: LCD port bitmap

The backlight is an independent bit: other communication should be done by setting up *then* enabling a command on the other bits. Bits in the port can be written to en masse or set/cleared selectively by writing to the appropriate port (fig. 3.9).

| Offset | Register | Writing ... | Reading ... |
|---|---|---|---|
| `00` | Data | All bits modified | Pin state |
| `04` | Not used | — | — |
| `08` | Clear | Any '`1`' bits written clear ('`0`') corresponding output bits | Port state |
| `0C` | Set | Any '`1`' bits written set ('`1`') corresponding output bits | Port state |

Table 3.9: LCD port register map

The *desired* output state of the eight data bits is always written and can be read back as the 'Port state'. However when `R/W̄` is '`1`' (read) the 'Pin state' may read differently. If the read operation is enabled (`E` = '`1`') then this will be a value supplied by the LCD module; if it is not enabled the value is *undefined*.

This port is *byte addressable* so it can be accessed in its entirety using word (or halfword) loads and stores or the control and data bits can be accessed separately using byte operations.

**Timer: base address `0001_0200`**

The timer consists of a 32-bit up counter with some accompanying control registers (table 3.10). The counter can be operated in various modes, according to the 'mode' programmed in the control register (fig. 3.11). When enabled the counter will increment every microsecond, wrapping around after $2^{32}$ steps or sooner in the limit/modulus register is enabled. Mode bits can be cleared or set selectively using a similar mechanism to the LCD interface.

| Offset | Register | Writing ... | Reading ... |
|---|---|---|---|
| `00` | Counter | Counter value (but generally don't!) | Counter value |
| `04` | Limit/modulus | Modulus - 1 | As written |
| `08` | Unused | — | — |
| `0C` | Control/status | Status bits | Status bits |
| `10` | Control clear | '`1`'s clear corresponding status bits | — |
| `14` | Control set | '`1`'s set corresponding status bits | — |

Table 3.10: Timer register map

When the counter counts to the value `0000_0000`, either by 'wrapping around' or reaching the value in the 'modulus' register if the modulus operation is enabled, bit 31 [6] is set in the control/status register. This bit is 'sticky' (p. 22): it remains set until cleared by:

- Explicit clearing by writing the bit to '`0`' or (preferably) clearing it using the control clear register.
- Writing to the counter or modulus register

This bit cannot be set directly by software. If enabled in the mode register, the terminal count status bit will assert an interrupt.

[6] Using the most significant (a.k.a. 'sign') bit is not uncommon in RISC-V systems: it is easy test for by comparing with `zero`.

| Bit | Function | Notes |
|---|---|---|
| 0 | Enable | '0' = halted; '1' = counting; |
| 1 | Modulus | Count modulo limit register |
| 2 | One shot | If '1', stop when count reaches 0 |
| 3 | Interrupt enable | '1' = interrupt output from bit 31; |
| 4 | Aux. clear count | Writing '1' here in clear register clears Terminal count |
| 5-30 | Not used | — |
| 31 | Terminal count | Sticky: set when count reaches 0; cleared by writing as '0' or by using mode clear register |

Table 3.11: Timer control/status register bits

**PIO: base address 0001_0300**

The Parallel Input/Output (PIO) interface is a general parallel interface. Each *bit* is independently programmable as an input, or as a digital output of values '0' or '1': there are therefore two register bits associated with each physical wire, one for its data and one for its direction. The associated port map is shown in figure 3.12.

| Offset | Register | Writing ... | Reading ... |
|---|---|---|---|
| 00 | Data | All bits modified | Pin state |
| 04 | Direction | Port bit direction 0 = Output, 1 = Input | Direction |
| 08 | Clear | Any '1' bits written clear ('0') corresponding output bits | Port state |
| 0C | Set | Any '1' bits written set ('1') corresponding output bits | Port state |

Table 3.12: PIO register map

The peripheral also has addresses for selectively clearing and setting bits in the data output register.

The **PIO is a 32-bit device** and thus should only be accessed using **word** loads and stores.

The 32 associated I/O lines are brought, in groups of 8, to the connectors on the front of the PCB and are used for external expansion. Each pin in the expansion connector can also be connected, on an individual basis, to another function (of the user's creation) using the 'Pin Function' bits in the 'System control' peripheral.

The correspondence of PIO bits to physical positions is shown in figure 3.2 and is also on printed on the PCB.


Figure 3.2: Expansion connection bit position in PIO

**Interrupt controller: base address 0001_0400**

The interrupt controller concentrates interrupt requests from diverse peripherals into the 'external' interrupt to the processor. It contains an accessible enable register for each of the possible 32 interrupt inputs so that the desired interrupts can be selected (fig. 3.13).

The 'raw' interrupt signals ("sources") can be read through this controller.

The ('external') interrupt controller concentrates the diverse set of peripheral interrupts into a single request for the processor. There are several software-accessible ports, primarily allowing *observation* of the hardware signals. The main *writeable* register provides individual interrupt enables for all the input signals; correspondence is bitwise, as would be expected. The results from after these AND gates is also readable for convenience ("requests").

| Offset | Register | Writing ... | Reading ... |
|---|---|---|---|
| 00 | Interrupt inputs | — | Interrupt input states |
| 04 | Interrupt enables | '1' = source enabled | Interrupt enables |
| 08 | Interrupt requests | — | Enabled requests in current mode |
| 0C | Interrupt mode | Select interrupt type | '0' = raw level; '1' = active edge |
| 10 | Interrupt edge (& clear) | '1's clear latched requests | Latched interrupt edge states |
| 14 | Interrupt edge set | '1's set latched requests | — |
| 18 | — | — | — |
| 1C | Interrupt output | — | Bit 0 reflects interrupt request output |

Table 3.13: Interrupt controller register map

Interrupt inputs may be *level-sensitive* or *edge-triggered*; this is selectable with the 'mode' register. A level-sensitive input relies on the interrupt request being cleared at its (peripheral) source while being serviced; edge-triggered interrupts each set a 'sticky' bit in a (software visible) register in the controller when they *become* active, which will remain '1' until deliberately cleared. Edge-triggered inputs are suitable for detecting that events *have happened*, which is appropriate in some cases. (It is also possible to set these bits from software – which will *request* the corresponding interrupt(s) from software; this is much less common/useful in practice but it can helpful in a few situations.) There are more details about this controller given in chapter 17 – particularly in figure 17.3.

For reference, figure 3.14 shows the implemented interrupt sorce positions: other bits are '0's.

| Bit(s) | Source | Function |
|---|---|---|
| 0-3 | User expansion | Customisable |
| 4 | Timer | Timer has reached its terminal count (at least once). |
| 5 | Button | Top left button has been pressed (at least once). |
| 6 | Video | Vertical sync. (pulse) |
| 7 | Video | (reserved) |
| 8 | Serial | The serial transmitter is available for writing. |
| 9 | Serial | The serial receiver has (at least one) byte waiting. |
| 10 | Serial | The serial receiver has an overrun error. |

Table 3.14: Interrupt input bit map

**Serial interface: base address 0001_0500**

There is a serial interface which allows bytes to be sent to/from a host terminal; this is effectively a simplified UART[7]: it lacks some of the control features of a full UART, such as bit rate control and error detection since here it is routed through the USB link.

A simple serial interface comprises a Transmitter (Tx) and a Receiver (Rx) which basically operate independently. Traditionally these have been combined in a small address space and the unit here reflects that (fig. 3.15). The registers here are only **8 bits wide** – so can exploit different size data transfers and are safe for any size load or store (but beware of LB sign extending a byte: LBU may be preferred).

| Offset | Register | Writing ... | Reading ... |
|---|---|---|---|
| 00 | Data register | Byte to send | Latest byte received |
| 04 | Status | Interrupt enables | Full status register |

Table 3.15: Serial interface register map

The *data* registers are separate but at the same address. Writing sets a value for transmission whereas reading returns the last received value. (This reflects the typical behaviour of such devices.)

[7] 'Universal Asynchronous Receiver/Transmitter': common name for a serial interface: look it up if you want details.

The status is reflected in a separate register (fig. 3.16): for both Tx and Rx there is a bit which indicates it is 'ready'. In the case of the transmitter this means that there is *space* to accept an output, so this will initially be *set*; the receiver is 'ready' when a byte has arrived so its status bit should initially be *clear*.

| Bit | Name | Function | R/W |
|---|---|---|---|
| 6 | RxERR | Overrun interrupt enable | R/W |
| 5 | RxIE | Receiver interrupt enable | R/W |
| 4 | TxIE | Transmitter interrupt enable | R/W |
| 3 | — | — | – |
| 2 | RxOVR | Receiver overrun | RO |
| 1 | RxRDY | Receiver ready | RO |
| 0 | TxRDY | Transmitter ready | RO |

Table 3.16: Serial interface status register

Writing transmit data will cause the transmitter to become unready for a time (until the byte is sent): thus writing should only be done when the transmitter is ready. The status change is automatic. Similarly the receiver ready status will be set once a byte has arrived; *this is* **cleared by reading** *the data.*

> *Note that if you* display *the serial data register via Bennett this will read the read and clear the status, so* – **beware!**

There are separate enable bits for both the transmitter and receiver interrupt status: note that the transmitter will be empty for much of the time, including following reset, so this should be disabled when not sending a message. One error indicator bit is present: this is the *Rx overrun*. Since there is no explicit flow control, if a received byte is not read within a certain time it can be overwritten a following byte: this is the 'overrun' condition and sets a 'sticky' status bit. The overrun status is cleared when the status register is read so if it is a concern then it needs to be checked each time this is done. The overrun can also be enabled to raise an error condition interrupt.

### VDU controller: base address `0001_0600`

The VDU controller will allow the display resolution to be controlled and the origin of the framestore memory (within the expansion SRAM) to be set. Some status information will be available.

It is currently incomplete and is not part of this module; it will be used in other modules in future.

### System control: base address `0001_0700`

This space is used for general device configuration and rarely used I/O. It contains some read-only system information – namely the version (date as three BCD fields) and the clock frequency in hertz – and the platform-level timer.

Another 32-bit register here redirects the PIO pins on the PCB to connect to user-defined I/O peripherals; a '0' bit here (the default value) connects the pin to the corresponding PIO bit: a '1' connects the pin to another I/O function, typically something added to the user expansion I/O area (section 3.5.3). This will be needed when chapter 21 is reached.

| Offset | Register | Writing ... | Reading ... |
|---|---|---|---|
| 00 | ID/Halt | Halt processor | System version |
| 04 | Clock speed | — | System clock (Hz) |
| 08 | Pin function | Register state '1' = alternate function | Register state |
| 0C | — | — | — |
| 10? | Time | RISC-V clock (low 32 bits) | RISC-V Clock (low 32 bits) |
| 14? | | RISC-V clock (high 32 bits) | RISC-V Clock (high 32 bits) |
| 18? | Time compare | Timer comparator (low 32 bits) | Timer comparator (low 32 bits) |
| 1C? | | Timer comparator (high 32 bits) | Timer comparator (high 32 bits) |

Table 3.17: System register map

### 3.5.3 I/O expansion: `0002_xxxx`

This 64 KiB memory-mapped area (in the privileged address space) is reserved for user-defined peripherals. The basic FPGA configuration has nothing here except 'stubs' to which signals can be connected. Signals can be connected to off-board devices if external pin-out is desired by connecting and programming the appropriate bits using the system control register (`0001_0700`: p. 21).

This area is used in exercise 8 (chapter 21) – and possibly exercise 9.

## 3.6 Hardware Development Environment

The CAD environment is **Questa**, which is probably already familiar. You should create a Questa project for this laboratory using the usual set-up script (i.e. "`mk_questa COMP22712`") and invoke it as appropriate for this laboratory (i.e. "start_questa COMP22712").

The hardware designs required in this laboratory will need to be integrated with software too.

For rapid, successful debugging use of **simulation is highly recommended**.

Fuller details are given when the appropriate exercise is reached (chapter 21).

**'Sticky' status bits**

*Not to be confused with the file system "sticky" property!*

This term is used for a flag bit which persists after being set until cleared by some other mechanism. A common example might be a flag indicating that some error has occurred which can then be checked at the end of a sequence of operations rather than at every step: it will reveal that at least one step (not necessarily the last) failed. One example might be on data transfers where a byte fails a parity check; another example could be an arithmetic overflow during some sequence where it is cheaper to verify an "okay" at the end of the operation than to check at each step.

In this module it is chiefly concerned with noting that some event has happened so that the ('momentary') event is not missed, can be picked up and deliberately cleared (acknowledged) by software.

**Testing Individual Bits in a Byte/Word: Masking**

Suppose you have a byte into which are packed the inputs from a set of eight switches. For convenience regard these as active high inputs with Button_0 corresponding to bit zero, Button_1 to bit 1, etc.

Thus, if Button_3 (only) is pressed, the byte (say, in a6) will be 0x08.

Let's say you want to test if Button_3 is pressed or not. One way to do this might be:

```
LI          t0, 0x08
BEQ         a6, t0, Button_3_pressed
```

Another sequence, close to the '`CMP`' (compare) function on many processors {x86, ARM etc.} might be:

```
SUBI        t0, a6, 0x08
BEQZ        t0, Button_3_pressed
```

This works if **only** Button_3 is pressed – but what happens if Button_1 (for example) is pressed at the same time?

How many comparisons would you need to ensure correct operation for any combination of buttons?

Many comparisons ("`if (...) ...`") are **arithmetic** functions (subtractions, in fact). In arithmetic functions there can be interactions between bits at different positions; this is useful in some circumstances, but not in others. On the other hand ("bitwise") **logical** functions treat all their bits individually. Logical functions are usually more appropriate when manipulating individual bit values.

This involves **masking** the bit(s) of interest; the result will be zero or not zero, which can be treated as Boolean states. On a RISC-V the code might be something like:

```
ANDI        t0, a6, 0x08
BNEZ        t0, Button_3_pressed
```

Notice that the RISC-V requires a 'temporary' register to hold the result of the masking operation unless you are prepared to lose the original value (`a6`, here).

You may other ways to do the same sort of test. For example:

```
SLLI        a6, a6, 28
BLTZ        a6, Button_3_pressed
```

(shift the bit of interest to the MSB and treat it as a sign bit) will work and preserves *some* of the other bits within `a6`. No doubt this is pleasing to some but it is somewhat obscure and would at least merit an explanatory comment!

# 4 | Programming Style & Practice

## 4.1 Structure

Some inexperienced "programmers" think that the choice of language affects the quality of the programming. They bandy terms like 'structure' and sneer that assembly language is, somehow, intrinsically 'unstructured'. One of the main arguments is that there are no "IF ... THEN ... ELSE"s and "REPEAT ... UNTIL"s, only the equivalent of "IF ... GOTO". These people will typically quote Dijkstra[1] as a reference. They are wrong!

To be fair to Dijkstra this is misapplication of his thesis; he cites the ability to jump randomly from place to place in a (high-level) programme as "an invitation to make a mess". *If* this is the case it is an invitation which should be politely declined. Here we get to the point of this section.

Programming is not a function of a language; it is much more than that. The majority of the art lies in analysing a problem and devising an algorithm which will solve that problem, which is implementable, and which is reasonably efficient. This is independent of the language chosen (or imposed).

The danger in assembly language programming is that it is much less restricted than operating in the confined syntax of a language; remember the language is compiled into machine instructions too! It can be much easier to write bad code in assembler than in many languages. It can also be easier to write good code. The point is that it is up to you to ensure that your code has an underlying structure.

There is not the time or space here for a course on structured programming. However here are a few tips:

- Use **procedures** (a.k.a. "functions", "subroutines", "methods") to isolate different identifiable operations. This allows a 'top down' 'divide and conquer' approach which is useful even if the procedure is only called once. In general procedures should communicate through passed **parameters**, which can travel both ways.

  Many high-level languages define functions which can have unlimited input parameters but only return a single quantity; this can encourage bad practice. In assembly language you can pass and return as many variables as you like.

  A procedure should have a well-defined entry and exit in the code. If you are being tempted to jump into the middle of a procedure take a cold bath and then modify your structure. The use of conditional returns can also be frowned upon, although they can prove a boon to efficiency.

- Keep careful control of the **scope** of variables. This is easy within a procedure — local variables are usually nestling in registers — but it requires self discipline not to just read or alter a variable in memory because you can. Resist! Data structures should be maintained by their own library of access procedures. This means that, when it is necessary to edit the code, changes can be handled locally. It is perfectly possible to write object-oriented assembly code if you choose.

- **Loop** structures should normally have a well-defined entry and exit point. Unlike most high level languages these may not be at the same point in the loop. If they are in the same place it is usual to enter at the top and exit at the bottom (when failing to close the loop back to the top again). If the entry and exit points differ it is usually more efficient to jump in and keep the exit point at the bottom[2].

[1] Edsger W. Dijkstra, "Go To Statement Considered Harmful", Communications of the ACM, Vol. 11, No. 3, March 1968, pp. 147-148. (http://www.acm.org/classics/oct95/ — go look it up, it's quite short.)

[2] The reason for this is left as an exercise.

- Beyond the obvious issue of the length of a data item assembly language does not have explicit **types**. It is therefore your job to ensure that your variables obey your own typing rules. It is very unusual to require floating point (float, real, ...) numbers in assembly language but integers, bytes, characters, pointers and more complex structures are common. An integer and a pointer may both be *represented* in a 32-bit word, but they are *different* things; for a start it makes sense to add integers, whereas adding addresses is nonsensical!

  You may choose to adopt a convention to help identify the type of items; for example prefixing **p**ointer names with "p_" or **c**haracters with "c_" makes them fairly obvious.

- **One 'thing'** belongs in (only) **one place**. It's a Bad Idea to evaluate anything in more than one code fragment; it causes all sorts of difficulties with 'maintenance'. If two things are the same – or *nearly* the same – then they should probably be integrated.
  This also applies to data.

## 4.2 Style

Although style is a personal thing, there are some practices which are worth observing to make your source code more readable, robust and maintainable. These are not all specific to assembly language either!

Firstly **layout**. You should be familiar with the general style of assembler source code, i.e:

```
Label           MNEMONIC  Addresses        ; Comment
```

It is a good idea to adhere to this format — the source code looks neat and labels stand out and therefore are easy to spot. It is also sensible to break up the code with blank lines to highlight groups of related statements (very approximately equivalent to a line of source code in a high-level language).

Within this framework it is possible to enhance readability in other ways. One obvious method is to choose **sensible label names** which reflect their function. As it is often difficult to think of many different names it is usual to name the *entry point* to a module (procedure etc.) sensibly and then use derivatives of this by suffixing numbers (or similar) to derive labels inside the module. (For example '`print_string`' might contain '`pnt_str_loop`'. This has the added bonus of indicating the extent of the module.

The comment field should be used to explain the meaning of the statement. It is usually possible to attach sensible comments to every line. An example of a nonsensical comment would be:

```
                ADDI       x9, x9, 1       ; Increment x9
```

however:

```
                ADDI       x9, x9, 1       ; Increment loop index
```

conveys significantly more information as it reminds us what `x9` currently holds.

The '1' in the preceding example is one of the few cases where it may be sensible to have immediate *numbers* appear in the address field. It is usually better to equate a number to a label and use that instead. This both conveys added information (the value 'FF' could be used for lots of things, but '`byte_mask`' is clear) and allows definitions to be changed in a single place (in a separate header file or at the top of the code) rather than trawling through looking for numerous references and, probably, missing some.

Finally, as the whole programme (being structured) is broken into procedures these can also be emphasised in the source code. Some more commenting can be added to document the procedure's function, its input and output parameters, any non-local data used, any registers which are corrupted etc. This is useful when trying to reuse or modify pieces of code, and it is the best place to keep this information in that it cannot easily become detached from the code itself.

### 4.2.1 Code example

```
; This routine prints a string, terminated by a '00' byte,
; pointed to by a0.  All register values are preserved.

Print_string    subi    sp, sp, 8       ; Push working registers
                sw      s0, 4[sp]       ;
                sw      ra, [sp]        ;
                mv      s0, a0          ; Argument => local variable

Print_str_loop  lbu     a0, [s0]        ; Load character (a0 not preserved)
                beqz    a0, Print_exit  ; Exit if terminator found
                addi    s0, s0, 1       ; Increment string pointer
                call    Print_char      ; Call character print
                j       Print_str_loop

Print_exit      lw      ra, [sp]        ; Pop working registers
                lw      s0, 4[sp]       ;
                addi    sp, sp, 8       ;
                ret                     ; Return
```

**Copy-and-Paste is Deprecated!**

Quite often, when programming, you discover that something you want to do is "rather like" something you've already done, with just a few differences. Probably your immediate reaction is to make a **copy** of the earlier code, **paste** it into the appropriate place and **edit** in the differences.

Don't do it!

If there is a lot of commonality between sections of code then they probably should be the same piece of code. This usually means some changes to the 'original' to accommodate some parameters or options and make it more flexible, often involving the extraction of a set of instructions into a procedure/function/subroutine/method/whatever. However the result is almost always a 'cleaner', better product.

**Avoiding Copy-and-Paste:**

**Advantages**

- **Readability** improves because, having understood a procedure, it can then be regarded as an abstracted function when it is encountered, rather than being deciphered (with subtle differences) each time.
  Improved documentation often results too - there's less to document!
- **Maintenance** is much easier. If (when) you are given a set of (someone else's) source files and required to "find the bug in there" you will welcome short, obvious procedures rather than 101 'variations on a theme'.
- Programme will be **smaller**. This may be vital in embedded controllers, where memory space is limited. Often speed increases too as a smaller memory image means fewer cache misses, page faults[a] etc.

**Disadvantages**

- The code may run slightly slower due to procedure call overheads.
  (This can be countered by using a **macro**[b] rather than a procedure.)
- Your productivity in number-of-lines-of-source-per-day falls significantly :-}
  (Productivity in terms of (better) products-faster improves though.)

[a] Don't worry if you don't know these terms; you will meet them in the future.

[b] A macro is a substitution where you can define your own keywords; each time a macro name is encountered the assembler (or compiler) substitutes the appropriate code.

**Optimising a `while` loop**

This example adapts the code in section 4.2.1 and adopts the ABI convention (section 3.3) for register use, and uses the relevant register aliases.

```
Print_string    subi    sp, sp, 8          ; Push working registers
                sw      s0, 4[sp]          ; Will be used as string pointer
                sw      ra, [sp]           ; Return address (another call below)
                mv      s0, a0             ; Move pointer to 'local' register
                j       Print_str_1        ; Start at 'test' point

Print_str_loop  call    Print_char         ; Call character print
                addi    s0, s0, 1          ; Increment string pointer
Print_str_1     lb      a0, [s0]           ; Load next character (as argument)
                bnez    a0, Print_str_loop ; Loop if terminator not found

                lw      ra, [sp]           ; Pop working registers
                lw      s0, 4[sp]          ;
                addi    sp, sp, 8          ;
                ret                        ; Return
```

A modern compiler will typically apply this form of optimisation. It's left to the (interested) reader to spot why this is (usually) faster.

**Optimising a `for` loop**

Note that in `C` the `for` (counting) loop is really a compacted syntax for a `while` loop. Here we are considering a 'counting' loop, although the principle applies to both.

```
for (i=0; i<100; i++) stuff(i);
```

May compile as:

```
                mv      s0, zero           ;
                li      s1, 100            ;
loop            mv      a0, s0             ; For sake of ABI; S0 is preserved
                call    stuff              ; Assume A0 is trashed
                addi    s0, s0, 1          ;
                blt     s0, s1, loop       ; Compare with terminal value
                ...
```

If you are simply iterating and *the ordering of the iterations does not matter* then **count down**.

```
for (i=99; i>=0; i--) stuff(i);
```

becomes

```
                li      s0, 99             ;
loop            mv      a0, s0             ; For sake of ABI; S0 is preserved
                call    stuff              ; Assume A0 is trashed
                subi    s0, s0, 1          ;
                bgez    s0, loop           ; Terminal value is zero
                ...
```

With RISC-V the saving is the use of a register `S1`, which will not then need explicit saving; on other processors (e.g. x86, ARM), which use status *flags* to control conditional branches the benefit is typically the omission of a '*compare*' instruction as there is a '*zero flag*' which will/can be set by the decrement. Whatever the compiler target, there is *likely* to be *some* advantage.

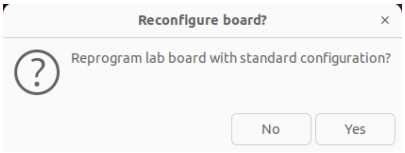
# 5 | Exercise 1: Simple Output

## Objectives

- Hardware/software familiarisation exercise
- Bit manipulation
- Simple I/O

## 5.1 Equipment

The laboratory equipment comes in two parts. First there is a software 'front end' which is provided by the Bennett software together with the RISC-V software development tools; *later on* we'll add the Questa/Xilinx hardware development tools too. Secondly there is a hardware 'back end' which is the FPGA board to contain the experimental computer with its embedded software and any daughter boards for specific, later exercises. These two parts are connected by a serial link which is carried across a USB link.




Ensure that the USB cable is in place and the power supply is connected. Open a shell window on the host computer and ensure the FPGA board is switched on. Type "**start_bennett_227**" (with an optional '`&`' suffix to get the prompt back) to start the interface. This should first enquire if you need to **configure the FPGA**: if you just switched on for the first time the answer will be '`Yes`'; if the board has been used previously it *may* already be configured. ('Yes' is safer; we will look at customised configurations in later exercises.) If the FPGA needs configuration it will take ∼20 s whilst the appropriate software starts and acts. Then the Bennett front-end should execute, the two parts of the system should synchronise and a window with various controls should appear; some red LEDs on the board (labelled '`UART`') should flash.[1]

The Bennett window (figure 5.1) has a number of areas which allow the memory and registers to be displayed and manipulated, the processor's execution to be controlled and programmes to be downloaded into both the memory and the FPGA. These are described below.

### 5.1.1 Manipulating registers & memory

Bennett has several independent panels which allow the display of memory and processor registers; these can also be 'popped out' as separate windows. A register panel displays a view of the RISC-V's register set, which can be displayed either from a 'hardware' number perspective {`x0, x1`...} or with the 'software' (ABI) names; the latter is probably easier to work with. Register values here may be editted.[2]

Memory views show a small area of the address space and can display the memory in various forms, primarily as different length hexadecimal numbers, ASCII and as disassembled RISC-V instructions. Hexadecimal is most useful for viewing data space whereas disassembly is convenient for following programmes. Extra (larger) memory windows can be created if needed. When modifying I/O locations (in particular) use the appropriate data size, both in Bennett and your own code.

[1] This is the serial traffic updating the front-end display.
[2] ... with the obvious exception of `x0`/'`zero`'

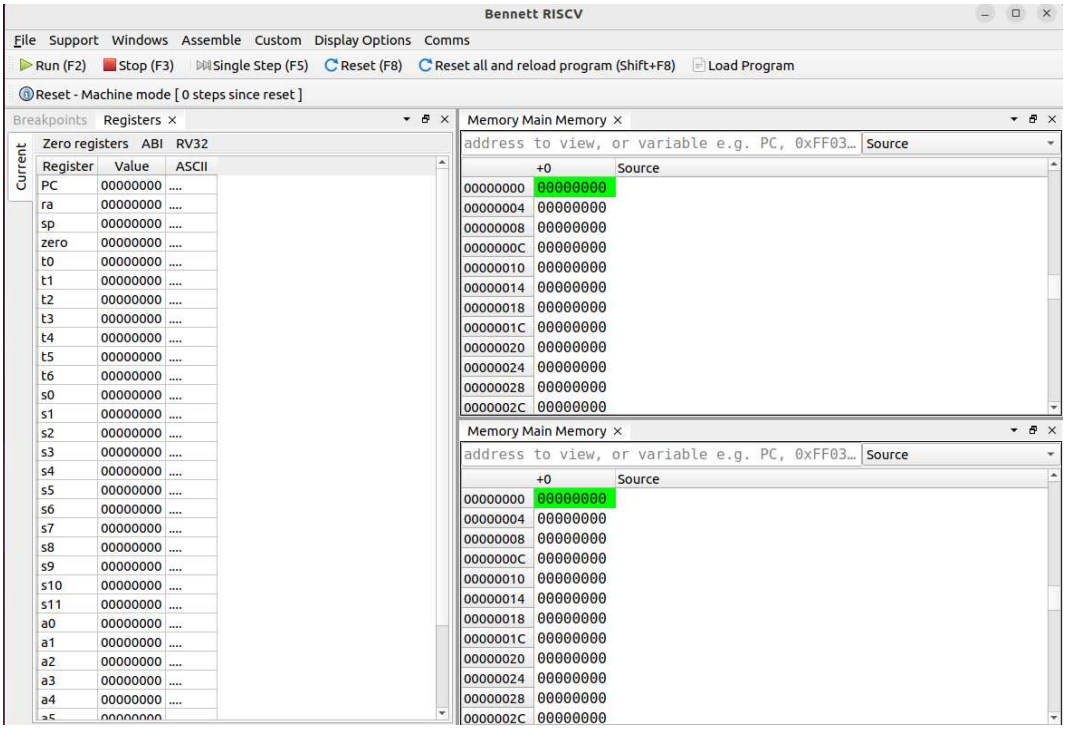



Figure 5.1: The Bennett front-end

It is possible to scroll the memory window around the address space, or move it to a particular address by typing this at the top of the window. It is also possible to define the window address in terms of a register's contents. For example, an expression such as "`PC-8`" will maintain a window which begins eight bytes (two instructions) before the current `PC` value — this is particularly useful in association with disassembly, as the window will then scroll as the programme executes.

Note: **all numbers are hexadecimal** and no prefices/suffices are required.

### 5.1.2 Developing code

There is no special text editor associated with this laboratory; develop your source files using your preferred tool. It is, however, recommended that you first create a directory to keep your files together. Assembler source files are usually identified with a "`.s`" extension. Don't forget to **save** the file before proceeding.

### 5.1.3 Assembling & Loading

There is a selection of assembly and loading options available via a pull-down menu. Successful assembly will result in the production of a `<filename>.kmd` file in your own directory; errors will produce appropriate error reports. This file is a human-readable 'list' file and may be useful as a reference, to see addresses etc. Loading can be automatic following a successful 'build'.

By default the file's load address will be `0000_0000`, but this may be altered for all, or part of the object code by using the `ORG` directive in the source file.

This utility only allows a single file to be assembled. Later it is possible that you may want to link several **source files** into a single **object programme**. One way of doing this is to 'include' other source files into the specified file using the directive:

```
INCLUDE <filename>      ; "GET" is synonymous
```

More sophisticated linking should not be needed in this lab. but if required can be done independently from Bennett using the almost any other tool you may choose, such as the GNU tools although this will make some of the 'source level' debug unavailable. The loader should recognise binary formats such as ELF.

### 5.1.4 Executing code

When the system is started the back end is in a 'Reset' state. This state can also be reached using the reset button on the Bennett window. When the RISC-V is reset the `PC` is set to `0000_0000` and the privilege mode (details later) is set to 'Machine' with interrupts disabled. Normally the processor would immediately begin executing (from address `0000_0000`) but in this case the processor is halted so that the state can be observed.

There are several ways in which a programme can be executed. The simplest is to use 'Run' which begins normal execution. This is suitable for debugged programmes but gives very little help if there are bugs present. Another mechanism is to use the 'single step' function, which executes a single instruction before halting the processor again. This makes it convenient to observe exactly what each instruction does. When single stepping it is often convenient to display a disassembly listing of the code in a memory window.

With longer programmes stepping each instruction individually can be tedious; Bennett also allows a programmable 'multi-step' (*** MULTI-FETCH? ***) facility which is useful, for example, in stepping through larger areas of debugged code. It is also possible to 'walk' through code by multi-stepping (could be set to '1') at a user definable step rate; automatically running a few instructions per second can make some tasks much easier to follow. A set of option buttons (which default to off) allow some compound operations to be considered as a single step. Initially the only useful ones will be `BL` and `SVC` which treat procedure and system calls as one step, respectively. Warning: these detect the end of the call by checking for the return address; although they should handle nested calls (including recursion) correctly a 'badly behaved' call (one which does not return to the expected address) will not terminate[3]. If this is encountered the processor will "run" and the 'stop' button will be needed to regain control.

Any of these operations can be stopped using the relevant on-screen button.

Bennett supports other methods of suspending execution which will be described later.

## 5.2 Input and Output Ports

To be at all useful a computer system must have some input and output (I/O) capability. The simplest I/O is provided by a **parallel port** which essentially maps a memory location into some real hardware. In the case of an output port this means that the state of the bits stored in this 'memory' are used to control some external hardware, such as some LEDs (Light Emitting Diodes). In the case of an input port the 'memory' is some outside world device — for example a single bit's state could come from a push button switch. The devices which inhabit this space are known as **peripherals**. On RISC-V systems these are '**memory-mapped**', i.e. they appear in the usual memory address space[4].

It is sometimes the case that peripheral devices do not use the full 32-bit bus. Some I/O devices in this laboratory have ports with fewer bits. On the other hand, some I/O devices may be simplified internally to only respond to certain sizes of data transfers. Check the appropriate specifications: for the equipment here there is a summary in table 3.7 (p. 17).

Note that when loading bytes (`LB`) and halfwords (`LH`) the RISC-V defaults to *sign extending* the values: `LBU` & `LHU` ('`U`'='unsigned') are often more appropriate operations.

### 5.2.1 Bit manipulation

Although I/O peripheral registers are byte wide or larger many inputs and outputs are smaller — often single bits. For example the position of a switch or button can be represented with a single bit. It is usual to cluster several (often *functionally* connected) I/O bits together into partial or full bytes to simplify the hardware requirements; an example is used in this exercise. Because the smallest quantity which can be addressed is (usually) a byte the issue of **bit addressing** must be handled in software. Bits are normally addressed such that bit 0 is the least significant bit.

[3] An example would be the "print following string" routine.

[4] This is not *always* true for all processors: e.g. historically the x86 architecture has a *separate* 'I/O space' although it is relatively small and not much used nowadays.

A 'trick' worth knowing is that any bit manipulation can be performed using two successive bit mask ANDed and XORed with the byte in question.

This is also useful when designing hardware.

| Action on bit | AND with ... | XOR with ... |
|---|---|---|
| None | 1 | 0 |
| Clear | 0 | 0 |
| Set | 0 | 1 |
| Toggle | 1 | 1 |

The RISC-V can alter a single byte in memory with a `SB` instruction. However if it needs to change a single bit it cannot modify it without, potentially, changing the other seven bits in the byte. To avoid this the other bits must be written back to their original value.

To do this the programmer must first find the original value of the byte. Usually this can be done by loading the byte, then altering the bit(s) concerned in a register. Individual bits can be set by ORing the value with the appropriate bit mask; bits may be cleared by ANDing with the (one's) complement of this mask. It is a simple and worthwhile exercise to learn the basic bit mask as shown in table 5.1.

| Bit Number | OR mask | AND mask |
|---|---|---|
| 7 | `80` (`1000_0000`) | `7F` (`0111_1111`) |
| 6 | `40` (`0100_0000`) | `BF` (`1011_1111`) |
| 5 | `20` (`0010_0000`) | `DF` (`1101_1111`) |
| 4 | `10` (`0001_0000`) | `EF` (`1110_1111`) |
| 3 | `08` (`0000_1000`) | `F7` (`1111_0111`) |
| 2 | `04` (`0000_0100`) | `FB` (`1111_1011`) |
| 1 | `02` (`0000_0010`) | `FD` (`1111_1101`) |
| 0 | `01` (`0000_0001`) | `FE` (`1111_1110`) |

Table 5.1: Hexadecimal (binary) bit masks

A bit can be 'toggled' (changed) — without knowing its initial value — by XORing it with a '`1`'.
Thus to set bit 5 of an 8-bit port the following sequence could be used:

```
LBU   t0, <port_address>
ORI   t0, t0, 0x20
SB    t0, <port_address>
```

(Remember that all RISC-V memory addressing must be relative to a register!)

In this first exercise a single 8-bit port is used for output. The bits within this port each correspond to an individual LED and are *active high* (i.e. a '1' turns the LED on). The port bits are read/write so the value written can always be read back.

| Bit | LED | |
|---|---|---|
| 0 | Red | (Left) |
| 1 | Amber | (Left) |
| 2 | Green | (Left) |
| 3 | Blue | (Left) |
| 4 | Red | (Right) |
| 5 | Amber | (Right) |
| 6 | Green | (Right) |
| 7 | Blue | (Right) |

Modern computers execute millions of instructions per second. It only takes a few instructions to change the state of a few LEDs. To make the exercise viewable by a human the steps must have 'reasonable' intervals between them. This can be done by using a **delay loop** — a construct which repeats wasting a little time many times over — but this is usually **bad** practice for several reasons[5].

## 5.3 Practical

Write a programme which cycles a set of 'traffic lights' attached to the output port at address `0001_0000`. Each light is controlled by a single active-high bit where `0` = light off, `1` = light on. The cycle should follow the same sequence which two sets of lights controlling a crossroads would do, and it should run continuously at a rate comfortably viewable by a lab. GTA (e.g. table 5.2).

[5]These should become apparent later.

| State | Left hand lights | Right hand lights | Bit pattern | Hold for about ... |
|---|---|---|---|---|
| 0 | Red | Red | | 1s |
| 1 | Red & Amber | Red | | 1s |
| 2 | Green | Red | | 3s |
| 3 | Amber | Red | | 1s |
| 4 | Red | Red | | 1s |
| 5 | Red | Red & Amber | | 1s |
| 6 | Red | Green | | 3s |
| 7 | Red | Amber | | 1s |

Table 5.2: 'Traffic light' sequence

(The bit patterns are left for you to fill in!)

Note that states #0 and #4 *appear* the same, but are actually different states.

### 5.3.1 Software delay loops

For this first experiment the timing reference *can* be a delay loop; however, to allow this to be upgraded later, you may choose to define a 'delay' procedure which can be called with an appropriate argument. A much more stable timing reference can be obtained by using a hardware timer which will be introduced in a later exercise.

The RISC-V in the laboratory has been set up with a clock rate of 40 MHz, which corresponds to a cycle time of 25 ns; *most* instructions execute in a single cycle although some (notably multiplication and division here) take significantly more cycles.[6] There is also a two cycle penalty for every flow control change {Jump, Call, Return, *taken* Branch} due to the pipeline *control dependency*. Thus, an empty loop (decrement counter & branch back) will iterate in 2 + 2 = 4 cycles or 100 ns; you can approximate your delays from this.

## 5.4 RISC-V Procedure Calls

Procedures are called using the `JAL` ("**J**ump **A**nd **L**ink") instruction although the pseudoinstruction `CALL` is more obvious. This branches to a specified label but saves the **return address** (i.e. the address of the following instruction) in a specified register. In the case of `CALL`, the default register `X1` – known as `RA` ('**R**eturn **A**ddress')[7] by the ABI is used automatically. Any existing contents of this register are lost.

To return from the procedure this saved value must be returned to the Programme Counter ("`PC`"). This is (really) done with a `JALR` instruction but it's *much* clearer to use the pseudoinstruction `RET` ("**Ret**urn") which simply copies `RA`/`X1` into `PC` without all the syntactic extras!

## 5.5 Advanced

Modify your programme to imitate a 'pelican'[8] pedestrian crossing, which has a different number of states and uses flashing lights to indicate some states.

| Bit | Label | Button |
|---|---|---|
| 0 (8) | SW1 | Top Left |
| 1 (9) | SW2 | Top Right |
| 2 (10) | SW3 | Bottom Left |
| 3 (11) | SW4 | Bottom Right |

Traffic lights are not always 'dumb' and may react to input stimuli. Modify your programme so that an input can 'trip' the lights to the appropriate state. Four input buttons are available towards the lower right of the circuit board for this function, which can be read as active high bits in a port at address `0001_0001` — or, if preferred as some higher bits if the LED port is read as a word (or halfword)at address `0001_0000`. Attempts to write to these bits will be ignored.

If attempting this remember that the inputs may change the traffic light timing, but not the sequence (which would be dangerous!) and that the button presses may be momentary[9] and on either (or both) buttons at any time. A modified state diagram might help here.

[6] Other complications due to dependencies following 'loads' et alia will also increase the time taken.
[7] It is sometimes called by the generic name "link register" too.
[8] Originally "pelicon" from **pe**destrian **li**ght **con**trolled.
[9] But you can't move your finger faster than the computer executes instructions!

**Hardware set & clear bits**

When dealing with I/O ports it is not uncommon to need to alter some bits in a register whilst others should be unaltered. One way to do this is via a read-modify-write operation: **load** a word, **modify** only the particular bits in the processor and **store** the result back.

This is sometimes adequate but, in certain circumstances with I/O devices, can fail. Consider if the hardware changes one of the 'uninvolved' bits between the load and store: the change is not seen by the processor and the store will undo it.

| Port | Port value | Store value | Result |
|---|---|---|---|
| Write | `0000_1111` | `0101_0101` | `0101_0101` |
| Clear | `0000_1111` | `0101_0101` | `0000_1010` |
| Set | `0000_1111` | `0101_0101` | `0101_1111` |

For such circumstances – and, sometimes just for convenience – bit changing operations may be provided. Typically these involve mapping an I/O register into several addresses where writing will have different effects.

A typical implementation (as depicted here) may have the ability to write *all* the bits or, selectively clear ('0') or set ('1') the bits where there is a '1' in the stored value, leaving others unchanged.

**RV32 immediate constants**

'Immediate' values are those which are embedded within the instruction stream. The basic RV32i uses 32-bit instructions and (obviously) not all of these can be used as the constant; other bits say what to do and where to put things. RV32 data operations can use **12-bit *signed*** constants: i.e. bit #11 is replicated in bits 31-12. This makes the values `FFFF_F800` to `FFFF_FFFF` and `0000_0000` to `0000_07FF` avaiable – the number range –2048 to +2047 in two's complement decimal.

The RV32 assembler provides a pseudoinstruction to allow easy access to arbitrary immediate numbers. The **L**oad **I**mmediate operation:

```
LI      Xd, expression
```

will assemble to an instruction, or a two-instruction sequence if necessary, which will load *any* 32-bit value.

**Loading addresses**

There are different ways to get an arbitrary (base) address into a register. Whilst all will produce the same result different pseudoinstructions are more relevant in certain circumstances.

For an address in a 'segment' *unrelated* to the code, `LI` is the most appropriate; this loads an *absolute* address.

For an address in the code 'segment' `LA` or `LLA` ('**L**oad **A**ddress' or '**L**oad **L**ocal **A**ddress') are more appropriate; the result is calculated as an *offset* from `PC` ("PC-relative") so provide relocatability.

> Why two different mnemonics? `LA` can be assembled differently in 'Position Independent Code' (PIC) using a 'Global Offset Table' (GOT) – which we don't do here.
> You can look up the terms if you're *really* interested.

**Accessing I/O ports**

All RISC-V memory addressing is made relative to a register. I/O is mapped into the memory space, thus a register is needed to point to the I/O port. Usually several different I/O ports inhabit a small area of memory.

Rather than loading a register with the exact address of the port each time a different port is used, consider pointing a register somewhere 'nearby', and using predefined offsets to get at the ports. For example:

```
Port_area       EQU     0x0001_0000
LED_port        EQU     0x0
Buttons         EQU     0x1
                ...
                LI      t0, Port_area
                LBU     s0, Buttons[t0]
                SB      s0, LED_port[t0]
```

This is faster and usually more legible, and uses fewer registers if several nearby ports are in use.

When dealing with I/O bits it is usually clearer to load bytes (and halfwords) as **U**nsigned quantities to avoid sign extension in the target register.

**I/O ports in C**

A 'problem' with programming in a more abstracted language is that sometimes you need to use I/O ports which are at *known*, fixed addresses. The usual solution to this in C is to set up pointers containing the relevant addresses explicitly. E.g.

```
const char* Port_LED = 0x00010000;     // (No '_' allowed here!)
```

This can then be used indirectly for access to the 'variable'.

```
*Port_LED = value;
```

If you have (for example) an I/O device with multiple registers this can be mapped (carefully!) to a `struct`; individual register names can then become the offsets ('fields', 'members' ... so many names for the same thing!) within that `struct`ure. This is helpful for readability and particularly useful when you have multiple instances of the same device.

**Let the assembler do the work ...**

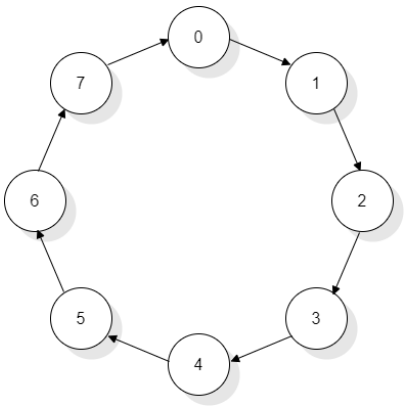
Some expressions are **static** — i.e. they can be evaluated at assembly ("compile") time. If this is the case they should be, saving instructions at run time (both time and space).
For example, to set two bits in a register, the bits can be combined by the assembler.:

```
ORRI     s6, s6, (bit_6 OR bit_1)
```

This looks more sensible when the labels are more informative names!
This example assumes that the result is representable in the limited immediate field, of course. For RV32 these are 12-bit *signed* values which largely restricts this to the *eleven* least significant bits. For some pseudoinstructions, notably `LI` (**L**oad **I**mmediate), there are fewer restrictions.

# 6 | Data-driven and Code-driven Algorithms

In the previous exercise you implemented a finite state machine in software.




You probably did this as a '**code-driven**' algorithm. This means that the current state is marked by the position in the code or, if you prefer, is implied by the value of `PC`. A code driven algorithm is illustrated below; this probably makes more decisions than your traffic lights.

```
state_0  do things
         IF inputs = 0 THEN GOTO state_1
         ELSE IF input = 1 THEN GOTO state_5
         ELSE GOTO state_2

state_1  do other things
         ...
```

An alternative method is to write a '**data-driven**' algorithm where the state is held in a state variable:

```
loop            CASE state
                0: CALL state_0_code
                   state := state_table_0[inputs]
                1: CALL state_1_code
                   state := state_table_1[inputs]
                2: ...
                GOTO loop

state_table_0   DEFW 1, 5, 2, ...
state_table_1   DEFW ...
```

In this second example one short piece of code covers any state machine and the behaviour is **looked up** in data tables.

The code-driven example is probably more intuitive and is easier to write — at first. It will probably also be shorter (no large data tables) and faster (no indirecting through memory).

The data-driven example is more regular and is thus easier to modify, expand and maintain.

Which is the "correct" solution depends on the task in hand and the programmer's own prejudices. They are presented here simply as alternatives for consideration.

## 6.1 Look-up Tables

A look-up table is a data table (array) placed in memory which can be indexed by an input variable. The state tables on the preceding page are examples of look-up tables.

Look-up tables are very useful for certain tasks. For example this would be by far the most efficient method of translating a 4-bit number into a bit pattern for a seven-segment display.

```
                ANDI    t0, a0, 0xF     ; Ensure input in range
                LA      t1, Seg_table   ; Point at table
                ADD     t0, t1, t0      ; Index desired element
                LBU     t0, [t0]        ; Load pattern
                ...

Seg_table       DEFB    0x3F, 0x06, 0x5B, 0x4F
                DEFB    0x66, 0x6D, 0x7D, 0x07
                DEFB    0x7F, 0x6F, 0x77, 0x7C
                DEFB    0x39, 0x5E, 0x79, 0x71
```

This is also quite fast because an arbitrary function can be calculated with a single memory load.

The look-up can be more complex than a single entry. For example a character set may be specified as a set of pixels with one byte (8 bits ⇒ 8 pixels) per line, and the character number would need to be multiplied[1] by the number of bytes in each character. For an 8×8 character matrix:

```
Char_set        DEFB    0x00, 0x00, 0x00, 0x00    ; Character 0
                DEFB    0x00, 0x00, 0x00, 0x00    ; (cont.)
                DEFB    0x80, 0x40, 0x20, 0x10    ; Character 1
                DEFB    0x08, 0x04, 0x02, 0x01    ; (cont.)
                ...
```

This may be useful if you want to explore further in the next exercise.

Look-up tables can also be useful in evaluating functions. For example "X/Y" — where X and Y are 8-bit numbers — can be looked up as:

```
                LBU     t0, x[a0]           ; a0 points at a variable 'struct'
                LBU     t1, y[a0]           ; 'x' and 'y' are offsets
                SLLI    t0, t0, 8           ; Multiply by 256
                ADD     t0, t0, t1          ; Calculate index
                LA      t1, Div_table       ; Point at table
                ADD     t0, t1, t0          ; Calculate address
                LB      t0, [t0]            ; Get result
```

This is fast, but expensive in memory requiring a $2^8 \times 2^8 = 2^{16} = 64$ KiB look-up table.

Worse, a 16-bit division would require 8 Gibytes ($2^{32}$ halfwords), more than the RV32 can address. Long division is therefore usually done iteratively. (The RISC-V used in this module includes the 'M' extension with the division instructions, anyway.)

[1] Easiest if the number is a power of two so the multiplication is simply a left shift.

# 7 | Exercise 2: Less Simple Output

## Objectives

- Parallel output/handshake sequencing
- Procedures and parameter passing
- A 'real-world' device

### 7.0.1 The Liquid Crystal Display (LCD)

Character-based liquid crystal displays (LCDs) contain some 'intelligence' and use a (reasonably) standard interface. The interface is a parallel bus interface which comprises data and control signals. The data bus can be 4-bits or 8-bits wide; the 4-bit mode is used when there is a very limited number of I/O signals available and need not concern us here. The interface signals are given in table 7.1.

The interface gives access to two 8-bit registers on-board the LCD controller. These are designated 'control' and 'data' and are selected by the state of the `RS` output. *Note: these registers are not directly visible to the processor.*

| Name | I/O | Function |
|---|---|---|
| DB[7:0] | I/O | 8-bit bidirectional data bus |
| R/$\overline{W}$ | O | Read not Write<br>0 = write, 1 = read |
| RS | O | Register select (address line)<br>0 = control, 1 = data |
| E | O | Enable<br>High to validate other signals |

Table 7.1: HD44780 Interface Signals

Like many I/O devices reading an LCD register may not give back the value last written to it. The type of operation performed is controlled by the R/$\overline{W}$ output.

To communicate with the display the control lines must be set to the appropriate values and then **strobed** using the enable line (fig. 7.1). If the enable signal is inactive the other signal states do not matter. During a transfer the signals should be stable, the usual constraints of set-up and hold times must be met, and the enable pulse has a minimum width.

When writing to an output port the bits will change approximately at the same time. To provide the correct timing the transfer will therefore require at least three separate output commands:

1. Set up data and command
2. Set enable line active (and pause long enough for the hardware to react)
3. Set enable line inactive

Note that it is not necessary to 'remove' the data after an output; it can remain until overwritten by the next command. However it may not be possible to complete step 1 with a single operation if, for example, only one eight bit port can be accessed at a given time (as there may be ten bits to change). In our system this can be done as one operation or with separate byte writes, whichever is neat/convenient.

The LCD controller is a HD44780[1], a standard part. It is smart enough to obey a small set of instructions. The simplest of these are:

- Print a character (and move cursor to next space)
- Clear the screen

[1] Actually the display fitted uses a newer, compatible part – a Sitronix ST7066U. This makes no difference here.

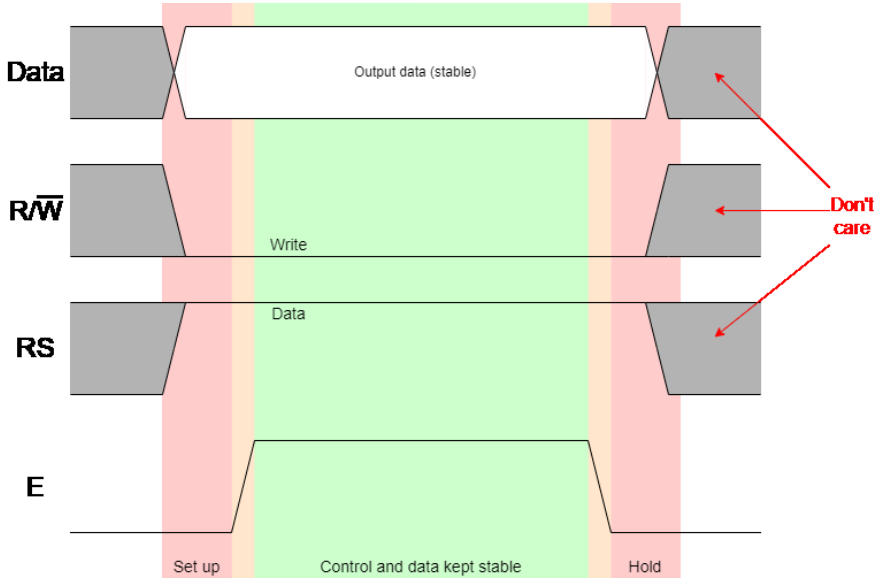



Figure 7.1: LCD data write cycle

A few other commands enable the cursor to be moved, the display to be scrolled and user defined characters to be downloaded. Data on the display controller may be found on the WWW if required[2].

The devices are quite slow, relative to the processor. This becomes apparent in two respects:

- There are minimum times between changes in the interface signals (fig. 7.1).
- Each command/write takes time to process and a subsequent command cannot be issued (successfully) until the current command is complete (fig. 7.2).

The first set of constraints are all quite short times and are most easily met by 'padding' with instructions in the issuing code. The relevant timings are:

- Command set-up time – small enough that separate output (store) operations will be adequate.
- Command hold time – small enough that separate output (store) operations will be adequate.
- Enable pulse width – min. 480 ns i.e. $\geq$20 processor cycles.
- Read data drive time – max. 320 ns i.e. $\geq$13 processor cycles.
- Enable pulse spacing – min. 1200 ns i.e. $\geq$48 processor cycles.

The last three need deliberate delay insertion (by software) and are large enough to merit a little delay loop (see p. 32). Most instructions take a single cycle (25 ns) and there is an extra 2 cycle penalty for every *taken* branch, so an *empty loop* takes 100 ns/iteration.

Each *command* takes a significant (and variable) time to process, from tens of $\mu$s to ~1.5 ms (thousands to tens of thousands of processor cycles). It is potentially quite easy to try to send the next character before the previous one has been printed: it will be ignored! As this will cause confusion it is important to wait until a write command is finished before sending the next. There are two ways of achieving this:

- Wait for at least the minimum *command* time.
- Wait until the display controller is not busy.

The first option is the less efficient because:

  - ◇ absolute times are relatively hard to measure.
  - ◇ the CPU could be doing something else for some (or all) of this time.

The second option is better because:

  - ◇ the CPU only waits as long as necessary.
  - ◇ it is self-adjusting to varying times for different commands, different LCD modules or different processor speeds.

[2]Try: https://www.sparkfun.com/datasheets/LCD/HD44780.pdf or find a site which suits you.

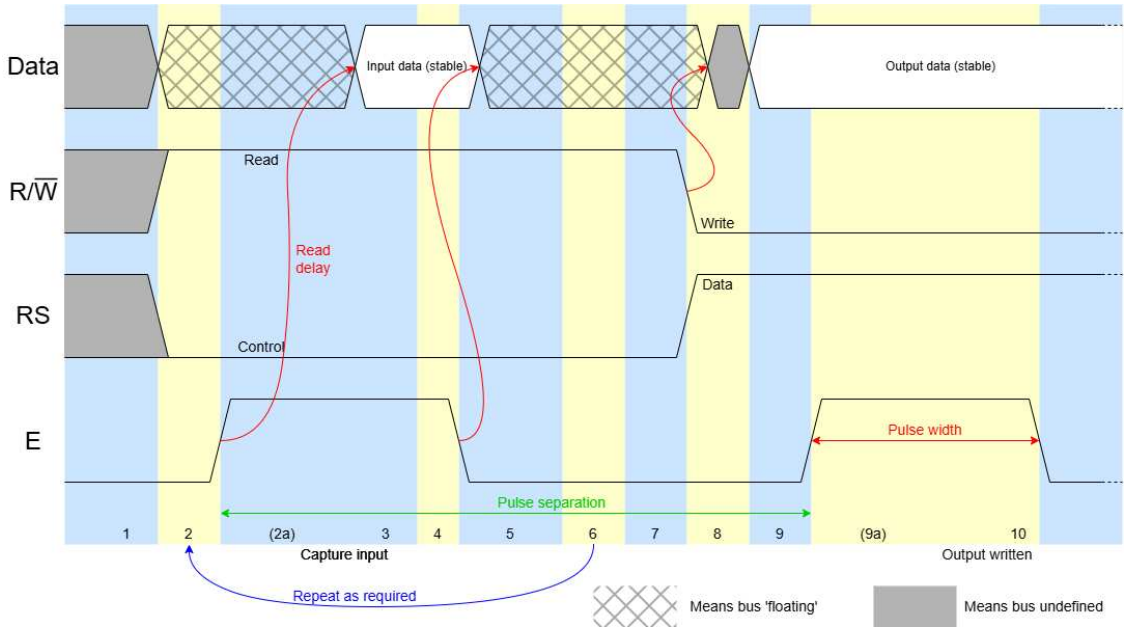



Figure 7.2: LCD character output sequence

However the second option (fig. 7.2) does require some feedback from the LCD. To obtain this the LCD control register must be read and bit 7 will then indicate the controller's status (0=idle, 1=busy).

To read the control register the appropriate *read* command must be sent, in this case `RS=0`, `R/W=1`. Before enabling the display the data bus — which will normally be an output — must be set to be an input. This then 'floats' the bus (i.e. it becomes tristate) so that, when it is enabled, the LCD controller may drive it. If this is not done two different values may be driven from each end of the bus which will lead to an undefined value, excessive power dissipation and, possibly, damage to the components. In our interface this protection has been built in.[3]

The sequence for writing a character now becomes:

| | | | |
|---|---|---|---|
| Wait until LCD controller is idle | 1 | Set to read 'control' with data bus direction as input {`R/W = 1, RS = 0`} | Control |
| | 2 | Enable bus (`E := 1`) | Control |
| | 2a | Delay to stretch pulse width | |
| | 3 | Read LCD status byte | Data |
| | 4 | Disable bus (`E := 0`) | Control |
| | 5 | Added delay to separate enable pulses | |
| | 6 | If bit 7 of status byte was high repeat from step #2 | |
| Write character | 7 | Set to write 'data' with data bus direction as output {`R/W = 0, RS = 1`} | Control |
| | 8 | Output desired byte onto data bus | Data |
| | 9 | Enable bus (`E := 1`) | Control |
| | 9a | Delay to stretch pulse width | |
| | 10 | Disable bus (`E := 0`) | Control |

The action of this on the interface signals is illustrated in figure 7.2. It all sounds like a lot of fiddling about, but each step is only one or two instructions. Once the routine is written it can be used to output all the characters you'll ever need.

Writing a control byte is similar — the only difference is the register addressed in step #7 (for the control register `RS=0`). The command for "clear screen" is `01`. There are plenty of WWW sites to help you find out about other commands if you want to do more; searching for "HD44780" is a good place to start.

[3]This issue will be revisited later.

| Address | Bit(s) | Byte [Bit(s)] | Direction | Signal(s) | Active state |
|---|---|---|---|---|---|
| 0001_0100 | 7-0 | 0 [7:0] | in/out | Data[7:0] | — |
| 0001_0101 | 8 | 1 [0] | out | LCD R/$\overline{W}$ | 0 = write to LCD module<br>1 = read from LCD module |
| | 9 | 1 [1] | out | LCD RS | 0 = control register<br>1 = data register |
| | 10 | 1 [2] | out | LCD E | 0 = interface inactive<br>1 = interface active |
| | 11 | 1 [3] | out | LCD Backlight | 0 = off<br>1 = on |

Table 7.2: LCD interface ports

## 7.1 The LCD Module Interface

The LCD uses eleven interface signals (plus the backlight switch which is an *independent* LED output): eight of these form the data port[4], the other three being control as shown in table 7.2. These are connected to two adjacent byte ports which may be accessed as individual bytes (`sb`, `lb`, `lbu`) or together at address `0001_0100` (`sh`, `sw`, `lh`, `lhu`, `lw`). Figure 7.3 shows the hardware configuration.

In this exercise the data port is *bidirectional* so there must be a means of disabling its output. The output enable for all eight data bits is connected to the direction control (bit 8: LCD_R/$\overline{W}$); this ensures that data can never be output at the same time as the LCD module is driving the signals.

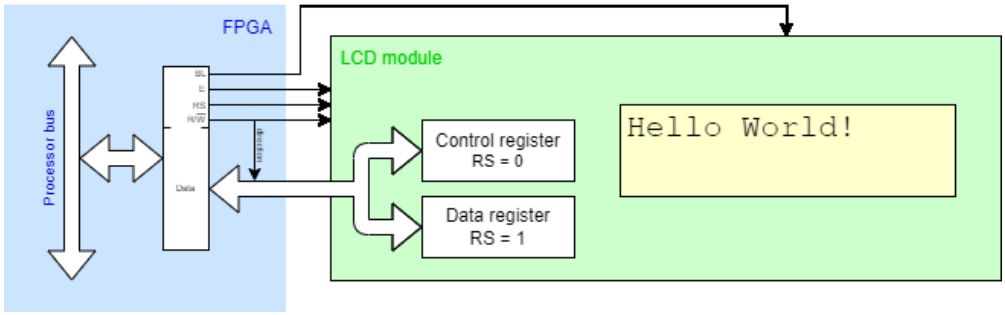



Figure 7.3: LCD controller interface

Reading LCD_data returns the values at the device pins. Thus reading this when the port is *enabled* as an output should return the value written to the port. If the LCD is *enabled* to read it will drive the signals; if the LCD is disabled the data will be floating and the value will be undefined.

When writing characters the LCD controller accepts standard ASCII printing characters (see page 113) with a few variants such as '¥' instead of '\'. A Japanese characters set and a few Greek/mathematical symbols are also included using codes from `A0-FF`; you may see some of these if something goes wrong! You can also define some of your own characters.

### 7.1.1 Bit clearing and setting

Many I/O ports have *bits* which provide independent functions packed together in bytes or words; an example here is the backlight, which is independent from the other LCD operations. The 'traditional' approach to altering bits within bytes/words is to load them into a processor, use bitwise logical operations {AND, OR, XOR} and store them back, as described in section 5.2. This will work okay and you can do this here.

[4]It is possible to programme the interface to use only four bit data. This saves signals (and the remaining ones fit neatly into a single 8-bit port) but means that two strobe cycles need to be made for every transfer.

Another approach is to have **hardware support** where selected bits can be changed whilst leaving others unaltered; this is available on this port by writing to the same bit positions at different addresses. This can sometimes make the (software) code shorter or clearer. The option is available on this port.

Occasionally this latter feature becomes genuinely important: this is where the hardware bits might change *autonomously* from the processor. This can introduce the 'classic' problem of '*atomicity*'where the processor's loaded value is out of date and inadvertently 'undoes' the hardware change when written back. This cannot occur on this port but the issue will return in a later exercise.

## 7.2 Practical

Write "Hello world!" (or some similarly trite message) onto the LCD display.

## 7.3 Suggestions

Write a routine to print a single character and call it repeatedly to print the string; the character may be passed as a parameter in a register. You will need this routine (or a derivative) later, so take care to comment the code.

The procedure for outputting characters is common to all characters. Outputting control and data is the *same process*, except for the state of one bit. It is sensible to combine these operations into a single procedure with an extra parameter.

## 7.4 Advanced

- Try outputting hexadecimal numbers using the same character output routine.
- You might like to try writing to the bottom line of the display.
- Try out some other features of the HD44780 controller, such as moving the cursor, scrolling the display, or even defining and displaying your own characters.
- Define some non-printing control characters which, when 'printed', have some effects on the display. This is like the '`\n`' you might see in (e.g.) C. Translation to control code(s) is required.

### 7.4.1 Commonly used control characters

Some common ASCII character functions are:

| Name | Byte | 'C' code | Commonly used to ... | HD44780 code(s) |
|---|---|---|---|---|
| Backspace (BS) | 08 | `\b` | Move the cursor left one place | |
| Horizontal Tabulate (HT) | 09 | `\t` | Move the cursor right one place | |
| Line Feed (LF) | 0A | `\n` | Move the cursor down one place | |
| Vertical Tabulate (VT) | 0B | `\v` | Move the cursor up one place | |
| Form Feed (FF) | 0C | `\f` | Clear Screen | |
| Carriage Return (CR) | 0D | `\r` | Move the cursor to start of line | |

Table 7.3: Common control characters

Note: writing these characters to the display will *not* achieve what you want. They must be **translated into codes or *sequences* of codes** which manipulate the HD44780 control register. However it is often very useful to be able to place (for example) a 'newline' code (such as '`\n`') directly into a string.

# 8 | Programming Hints

Code reuse is a very good idea; it saves you time and sweat in the future. However to make code easily reusable it has to be easy to modify, even when you've forgotten the details of how it works.

Here are some tips which will assist you in writing code now and will help you in future. This chapter is intended for reference so you may want to skip parts initially and comeback here later.

## 8.1 Setting Constants

Most programmes will contain various immediate constants. For example a string printing routine may detect certain 'control characters' which cause actions to happen rather than characters to appear. A common example might be "Carriage Return" ('`CR`') to start a new line.

> Actually Unix(-like) systems use "Line Feed" ('`LF`') whereas Windows/DOS has traditionally used both, i.e. '`CR`','`LF`' in combination.[1]

A subset of the ASCII character set (see page 113) is reserved for such functions although many of these are not now used for their original 'intended' purpose. '`CR`' is character `0x0D`; '`LF`' is character `0x0A`.

Many microprocessors (e.g. x86, ARM) will allow comparisons of register values with immediate values; RISC-V does not do this explicitly. Specifically, conditional branches are compare-and-branch, only comparing two registers. An arithmetic 'comparison' can be performed by using a 'spare' register.

```
SUBI    t0, s6, 0x0D        ; s6 is the working register
BEQZ    t0, it_matches      ; Compares with x0 == 0
```

although if a comparison is repeated it may be more efficient to put the comparison value into a register beforehand, *once*.

```
LI      t0, 0x0D
...
...
BEQ     s6, t0, it_matches
```

The statement:

```
SUBI    t0, s6, 0x0D
```

can check for such a character in a string. This works fine but can be a bit unclear when read. It is *much more obvious* to define a label at the start of a programme and use this consistently throughout.

```
c_CR            EQU     0x0D

                ...

                SUBI    t0, s6, c_CR
```

[1] Your assembler will interpret '`\n`' as a '`LF`' character.

**'Typing' variables**

Programming languages are more or less strict about the *typing* (interpretation of) variables. In assembly language there are just bytes, words etc. with no enforcement of their meaning.
You can sometimes make life easier by indicating the 'type' of a variable (or other symbol) with some convention, such as beginning character definitions with '`c_`', pointers with '`p_`' etc.
Just a suggestion ...

This makes the source code more readable, and distinguishes the '`0D`' used here from the same value used elsewhere for other purposes. Even better, use your own name for the constant. This allows easy modification if necessary; for example although many systems use '`CR`' to move to a new line, Unix/POSIX uses 'Line Feed' (`LF`). The excerpt below illustrates how this could be set up so that a single change can be used to modify all references in the code.

```
c_LF            EQU     0x0A
c_CR            EQU     0x0D
                ...
Newline         EQU     c_CR    ; Define appropriate character

                ...

                SUBI    t0, a0, Newline
                BEQZ    t0, Newline_routine
```

Note that, although the source code is longer the object code generated is the same in all these examples.

Search-and-replace with an editor is not a substitute for this. It is far too easy to modify something you didn't mean to.

## 8.2 Header Files

It is usual to have a large number of constant definitions at the start of a piece of source code. These may include items such as control characters, your own enumerations (e.g. "`TRUE`" and "`FALSE`"), processor specific features (e.g. "`Machine_Mode EQU &3`"), system specific constants (e.g. "`LED_port EQU 0x0001_0000`") and application specific numbers (e.g. "`Retry_count EQU 10`").

Many of these will remain the same for various projects and can be copied when starting a new source file. However such lists grow in size and this 'maintenance' may well need applying to all the source files. Therefore it is usually sensible to keep the majority of constants (all except the application specific ones) in a separate file which can be included at assembly time.

In the RISC-V assembler these two directives are synonymous:

```
INCLUDE filename.s
GET     filename.s
```

In practice, including more than one header may be sensible: for example, one could contain items related to the application while another has system information (such as the memory size and I/O port locations) for the board it's run on. That way the code can be ported to a different system simply by changing a single inclusion name.

It may also be sensible to split your projects into more than one file. A sensible split (in some suture exercises) is to divide any 'system' code — which would encompass the more reusable 'OS' code — from the user-mode applications code. Later, these will occupy different regions of memory.

**Numeric constants**

Bennett displays everything primarily in hexadecimal; this is typical in a debugger.

Like most development tools the default base (or "radix") for numbers in the assembler is 10, i.e. "decimal". This is because it is the default for (most) people. However, in your source code you should use **whichever base makes your code most readable**: sometimes this will be **decimal** ("`250`"), sometimes it will be **hexadecimal** ("`0xF0`": a '`$`' prefix is also acceptable) and, sometimes, **binary** ("`0b11110000`" or "`:1111_0000`").

The default assembler allows you to intersperse '_' characters in numbers to make them more readable if there are a lot of digits.

## 8.3 Multi-way Branches

A common construct in programming is the so called "`case`" or "`switch`" statement which is a multi-way branch depending on some variable or expression. This is also useful in assembly language, where it is normally implemented as a jump table. An example in RISC-V code, with the index in a0, is given below:

```
                LI      t0, Table_size          ; Range check first
                BGEU    a0, t0, Out_of_range    ; Using 'Unsigned' forces 0 minimum
                LA      t0, Jump_table          ; Point at table
                SLLI    a0, a0, 2               ; Multiply to index words
                ADD     t0, t0, a0              ; Calculate table entry address
                LW      t0, [t0]                ; Load target address from table
                JR      t0                      ; Jump

Jump_table      DEFW    Routine_0
                DEFW    Routine_1
                DEFW    Routine_2
                DEFW    Routine_3
                DEFW    ...
```

Note that:

- the offset (in `a0`) is multiplied by 4 (<< 2) when indexing the table to allow for the addresses in the table being four bytes long.
- this code jumps to the routines which must know where to jump back to. To change these into procedure calls ("`CALL`") it is necessary to save a 'return address' (`ra`/`x1` by convention) before the jump, e.g. `LA ra, Place_to_return_to_afterwards`

Note that the index is **range checked** for legality before being looked up in the table; picking up an 'address' outside the defined range will result in an instant crash!

Such a code sequence will later become useful in, for example, sorting out traps (p. 63) or **dispatching** different supervisor calls to their appropriate **service routines** (p. 71).

**Komodo tip**

It's possible to display labels instead of addresses in a Komodo memory window; it's also possible to use labels as part of an *expression*. This can make it much easier to find/display particular memory locations of interest.

## 8.4 Allocating Variables

Variables can be held in registers or in memory. The act of 'declaring' a variable is necessary to reserve some space in memory to hold that particular value. The easiest way to reserve space in assembly language is simply to 'define' the relevant item (usually a 32-bit word) in the source file using "`DEFW`". Typically — as the value is initially unknown — the word would be defined to be zero, but this is arbitrary.

```
                LA      s6, thingy      ; Get address
                LW      t5, 0[s6]       ; Load variable
                ADDI    t5, t5, 1       ; increment
                SW      t5, 0[s6]       ; and store back
                ...
thingy          DEFW    0               ; Arbitrary variable
```

This can be quite tedious: often a register is dedicated to point to such variable space. Having a register dedicated to each variable would use a lot of registers: instead it is efficient to use address offsets (explicitly included as '`0`' in the above example.

The assembler has a mechanism to define **offsets** rather than absolute labels, as follows:

```
                LA      s4, variables   ; Get address
                LW      t5, wotsit[s4]  ; Load variable
                ADDI    t5, t5, 1       ; increment
                SW      t5, wotsit[s4]  ; and store back
                ...
variables       struct
thingy          word                    ; Arbitrary variable
wotsit          word
                ...
```

Here '`variables`' is an ordinary label and will be set to the address in memory; however '`thingy`' is an *offset* in the structure, intended to hold a `word` and will be set to '`000`'. '`wotsit`' is another offset, a word (4 bytes) later so has the value '`004`' and so forth. (See p. 108 for more details.)

**RISC-V load/store 'global' pseudoinstructions**

As well as the standard `LW` and `SW` (true) instructions – and all their sized and unsigned variants – there are some pseudoinstructions which share the mnemonics. These can be used for access to *any* address but assemble to two real instructions, so they can be slower. There is also some syntactic restriction.

```
                LW      S0, label_0     ;
                LBU     S1, label_1     ; etc.
```

will load from the memory at the label (using PC-relative addressing).

Similarly:

```
                SW      S2, label_2,  t0 ;
                SB      S3, label_3,  t0 ; etc.
```

will store the first register's contents to the memory at the label (using PC-relative addressing) but needs another, *different* register specifying to act as as a base address in the process. This temporary register (`t0`, above) is best regarded as 'undefined' after these operations.

Pseudoinstruction translations are given in table 3.3 on page 9.

**Breakpoints**

A breakpoint is a 'marked' instruction which stops execution when it is encountered. Execution is stopped *before* the instruction can change the state of the system. They are used as a debugging aid; for example a breakpoint set just before a suspect piece of code allows the programme to be 'run' to that point and then single stepped. Alternatively a breakpoint could be used to stop execution if the processor has jumped to an address it should not have reached.

The RISC-V boards support up to 8 breakpoints and each has three possible states:

- deleted (or never set).
- active (can cause a trap if encountered) — highlighted in yellow.
- inactive (will not cause a trap but the programmed values are retained) — light grey.

In addition a *global* flag (active by default) must be set before *any* breakpoints are considered.

The easiest method of adding/removing a breakpoint is to 'double click' on an address in a memory window, which will toggle the state. (They can also be controlled from a pop-up window.)

Note that a breakpoint is ineffective if it is the first instruction encountered after a user 'run' command; this allows the user to elect to 'continue' from a breakpoint without it causing interference unless it is encountered again.

**Watchpoints**

A watchpoint is similar to a breakpoint (p. 46) in that it is a 'marked' *data* location/area which stops execution when it is referenced. Execution is stopped *before* the instruction can change the state of the system. As a debugging aid they can be used (for example) to see what may be causing changes to a variable.

**In this module . . .**

The RISC-V boards support up to four watchpoints. For flexibility, these can cover *areas* of memory so that (for example) an entire data structure or the whole stack area could be monitored. This is done by applying a binary **mask** to the address. For example, if a watchpoint address was set to `0000_0123` and the mask to to `FFFF_FF00` then any reference with a base address `0000_0100` - `0000_01FF` would be included.

Each watchpoint can also be independently enabled for loads and stores, so can be set to detect load, store, both or neither (i.e. disabled). More status bits are present to allow each transfer **size** {byte, halfword, word} to be enabled independently and the **privilege** mode of the operation can be filtered too.

In addition a global flag must be set before any watchpoints are considered.

The easiest method of setting/clearing a watchpoint is . . . WHAT?? . . . if there is one.

Note that a watchpoint is ineffective if it is the first instruction encountered after a user 'run' command; this allows the user to elect to 'continue' from a watchpoint without it causing interference unless it is encountered again.

# 9 | Stacks: "Why?" ... and "How?"

*This section should be revision!*

## 9.1 Procedure Calls

We have already mentioned how a procedure can be called using the "`CALL`" (or "`JAL`") instruction: this is similar to a branch but it leaves a 'return address' in a register. The RISC-V *convention* – assumed in `CALL`, optionally explicit in "`JAL`" – is to use `ra`, a.k.a. `ra`.

The branch can therefore be 'undone' by moving this value back into the `PC`, either explicitly with `JALR ra` or implicitly with the `RET` pseudoinstruction. This behaviour is illustrated in the following (rather pointless) code fragment.

```
11B8    ...
11BC    ...
11C0    ADD     s0, s6, s7      ;
11C4    LI      a0, 6           ;
11C8    CALL    $1234           ; Leaves ra = 11D0
11D0    SUB     s0, s0, a0      ;
11D4    LI      a0, -1          ;
11D8    CALL    $1234           ; Leaves ra = 11DC
11DC    ADD     s1, s1, a0      ;
11C0    ...
        ...
1230    ...
1234    SLLI    a0, a0, 1       ;
1238    RET                     ; Return
123C    ...
```

An **execution trace** of this code would show the instructions executed as follows:

```
11BC    ...
11C0    ADD     s0, s6, s7      ;
11C4    LI      a0, 6           ;
11C8    CALL    $1234           ; Leaves ra = 11D0
1234    SLLI    a0, a0, 1       ;
1238    RET                     ; Return (to ra)
11D0    SUB     s0, s0, a0      ;
11D4    LI      a0, -1          ;
11D8    CALL    $1234           ; Leaves ra = 11DC
1234    SLLI    a0, a0, 1       ;
1238    RET                     ; Return (to ra)
11DC    ADD     s1, s1, a0      ;
11C0    ...
```

Notice that the procedure (or "**subroutine**") has been 'inserted' twice, at different points in the parent code.

Also notice that a parameter has been passed (in `a0`) which is different for the two invocations and a result is also returned (also in `a0`). Procedures which return values are often called "**functions**". An example of parameter passing would be a character to a 'print' routine.

## 9.2 Stacks

A potential problem with the RISC-V's procedure calling mechanism is that the `CALL` instruction simply *overwrites* the contents of `ra`. This means that the called procedure cannot itself call another procedure (a process known as "**nesting**") without overwriting this value.

Nesting procedures is extremely useful. To enable it the return address must first be copied from `ra` to somewhere where it will be preserved, usually a memory location; the **stack** is convenient.

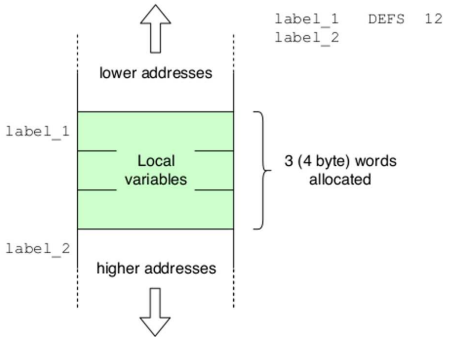
A stack is a data structure which is used for temporary variables. It is a "last-in, first-out" (LIFO) structure where (in principle) only the top item is available A stack is just a data structure and you can build as many as you want. However they are so useful that there is often support in a processor's instruction set architecture (ISA) to help with stack operations.

In principle, when an item is stacked all the other elements move one 'slot' further away, and they all move back one 'slot'. In practice moving data about would be hopelessly inefficient; instead the data remain stationary and the address of the top of the stack moves. This address therefore needs to be maintained and it is kept in a **stack pointer**. Typical systems only dedicate a single register to this function and to the same stack will be used for several functions, usually just known as 'the stack'. The RISC-V convention is to use `x2` as the default stack pointer and "`SP`" is an alias for this register.

Of course before it is used the stack pointer must be set up at the 'start' of a free area of memory. This will typically be done as part of the system initialisation. A programme which is running doesn't care what value the address in its stack pointer is.

**Stack set-up**

A stack requires an area of memory and a stack pointer (`SP`). In assembly language a *space* in memory may be reserved using a `DEFS` (**DEF**ine **S**pace) directive: an argument gives the size in *bytes*. (Note that stack operations will operate on *words* so the size should be a multiple of 4.)




If the directive is *labelled* the label will be the first (lowest address) byte of the reserved space. Note that – by convention – stacks grow *down* in their address space; thus `SP` should be initialised to the 'end' of the space or – since the `SP` is typically *pre*decremented – the address immediately following the space. The `SP` should always be kept **word aligned**.

(In the RISC-V *calling convention* the stack pointer must be 16-byte aligned!)

**Stacking in RISC-V**

RISC-V does not have specific `PUSH` & `POP` instructions: the operations must be built 'by hand' by adjusting `SP` and storing/loading.

Although these operations may be done in either order it is recommended that `SP` is altered *before* storing and *after* loading. This means there is never valid data at the 'wrong side' of `SP`.

**Stack terminology**

A "**Full**" stack is one where the Stack Pointer holds the address of the last valid value; in an "**Empty**" stack the Stack Pointer holds the address of the first (next) unused location.

A "**Descending**" stack grows from higher to lower numbered addresses; an "**Ascending**" stack grows the other way.

'Full Descending' ('**FD**') is the most commonly encountered stack convention and is used in the RISC-V ABI.

**Recursion**

Stacking is particularly convenient because it allows a procedure to call itself. This process — known as "**recursion**" — is sufficiently useful to allow for. (Naturally there must be some condition to terminate this!)

For the 'classic' example of recursion look up "Tower of Hanoi" (sometimes known as the "Tower of Brahma").

**Source code vs. disassembly**

Source code specifies what the programmer wants – or, at least, wrote! The processor has only the memory image, which may *appear* different when *disassembled* back into assembly code. Labels will have disappeared and some pseudoinstructions may not be deduced, as shown by the example below.

Source code (with list file):

```
000011B8: ........   ;               ...     .....
000011BC: ........   ;               ...     .....
000011C0: 007302B3   ;               ADD     s0, s6, s7      ;
000011C4: 00600513   ;               LI      a0, 6           ;
000011C8: 06C000EF   ;               CALL    routine         ; Leaves ra = 11CC
000011CC: 40A282B3   ;               SUB     s0, s0, a0      ;
000011D0: FFF00513   ;               LI      a0, -1          ;
000011D4: 060000EF   ;               CALL    routine         ; Leaves ra = 11D8
000011D8: 00A585B3   ;               ADD     s1, s1, a0    ;
000011DC: ........   ;               ...     .....
          ........                   ...     .....
00001230: ........   ;               ...     .....
00001234: 00151513   ; routine       SLLI    a0, a0, 1       ;
00001238: 00008067   ;               RET                     ; Return
0000123C: ........   ;               ...     .....
```

Disassembly:

```
000011B8: ........   ;               ...     .....
000011BC: ........   ;               ...     .....
000011C0: 007302B3   ;               ADD     s0, s6, s7      ;
000011C4: 00600513   ;               ADDI    a0, x0, 6       ;
000011C8: 06C000EF   ;               JAL     ra, $1234       ; Leaves ra = 11CC
000011CC: 40A282B3   ;               SUB     s0, s0, a0      ;
000011D0: FFF00513   ;               ADDI    a0, x0, -1      ;
000011D4: 060000EF   ;               JAL     ra, $1234       ; Leaves ra = 11D8
000011D8: 00A585B3   ;               ADD     s1, s1, a0    ;
000011DC: ........   ;               ...     .....
          ........                   ...     .....
00001230: ........   ;               ...     .....
00001234: 00151513   ; routine       SLLI    a0, a0, 1       ;
00001238: 00008067   ;               JALR    x0, [ra]        ; Return
0000123C: ........   ;               ...     .....
```

Note the translation of:

- `LI` to `ADDI` exploiting `x0`: `LI` can become a two instruction sequence if the value exceeds the simply representable immediate range {`0xFFFF_F800`:`0x0000_07FF`} ({-2048:+2047}).
- `CALL` to `JAL`: `CALL` can become a two instruction sequence {`AUIPC`, `JALR`} if the target is more distant.
- `RET` to `JALR`

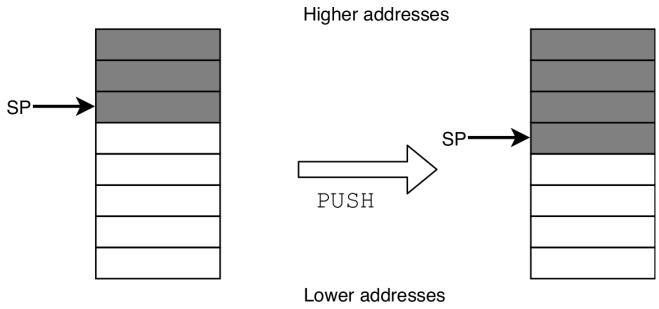
It can also be observed that the *object code* for the two, apparently identical, `CALL` operations differs because the target is the same but their own position (and hence their *offset*) is different.

N.B. Some of these pseudoinstruction optimisations may differ with other assemblers.

### 9.2.1 Stack use

This maybe in next chapter

There are different ways to implement a stack; it is recommended (though not compulsory) that the stack model known as "full descending" is used since this is the commonest one encountered. In this model the Stack Pointer (SP) contains the address of the last item pushed, and subsequent 'pushes' store data in lower-numbered addresses.




The act of adding an item to a stack is usually called a "stack push" – or just a "push"; the act of removing an item is called a "stack pop" – or just a "pop"[1].

With this type of stack the act of pushing (say) s0 would become:

```
SUBI    sp, sp, 4
SW      s0, [sp]
```

i.e. SP is decremented before the data transfer.

The complementary 'pop' operation would be:

```
LW      s0, [sp]
ADDI    sp, sp, 4
```

i.e. the load occurs from the current SP address which is then incremented.

In typical code it is quite common to wish to push/pop several registers at the same time. This can be done individually but (of course) it is more efficient to use address offsets.

```
SUBI    sp, sp, 12
SW      s2, 8[sp]
SW      s1, 4[sp]
SW      s0, [sp]
...
LW      s0, [sp]
LW      s1, 4[sp]
LW      s2, 8[sp]
ADDI    sp, sp, 12
```

The implication with a "full descending" stack is that the stack pointer should be initialised to the address *immediately 'above'* the area reserved for the stack; the first stack push will then predecrement the address into the allocated region. For example if addresses up to 0000_3FFF are allocated for a stack the (full descending) SP should be initialised to 0000_4000 so the first word will be pushed to 0000_3FFC[2].

If another, nested procedure is to be called then it is important that the return address is preserved, usually by pushing. It is also likely that a procedure needs some "scratch" registers for its private workspace. To create space it is possible to push the original contents of these registers on entry to the procedure at the same time as the return address and pop them back at the end. The choice of convention is up to the individual programmer but attention is drawn to the standard ABI (section 3.3) which specifies some registers which are not expected to be preserved.

[1] "Pull" has been used but is uncommon.
[2] I.e. a *word* address

# 10 | Exercise 3: Nesting Procedure Calls

## Objectives

- Revise the RISC-V call/return mechanism
- Set up a process stack

## 10.1 Stack Operations on a RISC-V Processor

It is possible to build stacks in a number of different ways. It is important to know which model is in use because — unlike many processors — there is no explicit "`PUSH`" or "`POP`" operation; instead these are done using load and store operations combined with a register adjustment.

It is recommended (though not compulsory) that the stack model known as "full descending" is used (figure 10.1). In this model the Stack Pointer (`SP`) contains the address of the last item pushed, and subsequent 'pushes' store data in lower-numbered addresses. This is the most widely used convention.

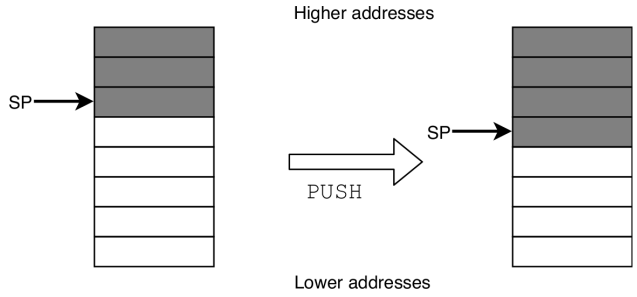



Figure 10.1: Stack push: before and after (Full Descending stack model)

In this mode the operation "`PUSH S0`" becomes:

```
SUBI    SP, SP, 4
SW      S0, [SP]
```

i.e. `SP` is decremented before the data transfer.

**Push and Pop**

This is computer jargon.

- "Push" means 'add a quantity to the stack'
- "Pop" means 'remove a quantity to the stack' (occasionally called "pull")
- The values on the stack do not move; instead the **stack pointer** (usually `x2` – a.k.a. '`SP`') changes to indicate the new position of the '**top of stack**'.

Conversely the "`POP  S0`" operation would be:

```
LW      S0, [SP]
ADD     SP, SP, 4
```

The code fragments above could be written in a different order, using (negative) offsets on the memory transfers. Although it would be safe in these circumstances it is generally **good practice** to assume that everything at the 'wrong' side of the stack pointer is not defined and 'at risk'. This can be important on some processors – even, potentially, on RISC-V – if the stack is also in use for handling interrupt servicing which could overwrite anything placed there.

In typical code it is quite common to wish to push/pop several registers at the same time. This can be done reasonably efficiently by exploiting offsets.

```
SUBI    SP, SP, 12      ; Three 4-byte words
SW      S2, 8[SP]
SW      S1, 4[SP]
SW      S0, [SP]
...
...
LW      S0, [SP]
LW      S1, 4[SP]
LW      S2, 8[SP]
ADDI    SP, SP, 12
```

## 10.2 Procedure Returns in RISC-V Code

If a procedure is a 'leaf' procedure[1] which calls no other procedures then the return address may be left in the link register (`ra`/`x1`).

If a procedure calls one or more other procedures then the return address must be preserved preferable by stacking ("pushing") the return address. A typical instruction to do this would be:

```
SUBI    SP, SP, 4
SW      RA, [SP]
```

If this is done the value must later be recovered by unstacking ("popping") the value in the corresponding manner:

```
LW      RA, [SP]
ADDI    SP, SP, 4
```

Of course, other registers may be (and often are) saved/restored at the same time. More information about a software convention for this may be found in section 16.2.

This return itself is accomplished by moving the popped register to the PC. The instruction for this 'in full' is:

```
JALR    ZERO, [RA]
```

but this is more conveniently (and more clearly) shortened to the pseudoinstruction:

```
RET
```

[1] i.e. a furthest point in the 'call tree'.

## 10.3 Practical

Structure your solution to exercise 2 so that it contains two subroutines, one of which prints single characters and the other which prints a string. The string should be defined by a pointer (i.e. an address) passed into the latter routine and should be output by calling the print character routine repeatedly.

Use these routines to print **two** (or more) different strings in the same programme; that way it's easier to see it is returning correctly.

The routines should be callable by a user who knows nothing about their contents — i.e. they should be utilities. This means that, when called, they should leave all registers unchanged and only "do what they say on the can", i.e. print a character or string.

Don't forget to allocate some memory for the stack and initialise the stack pointer before using it. Memory can be allocated using (for example):

```
DEFW    0, 0, 0, 0      ; Reserve four words
```

or, more practically:

```
DEFS    100             ; Reserve 100 bytes
```

(Note that this reserves bytes, not words.)

## 10.4 Advanced

Try one or more of the following:

Print different strings on different lines of the display.

Print "top left", "bottom right" etc. as a response to pressing the buttons on the board.

## 10.5 Submission

You should submit this exercise for assessment and feedback. The submission should include code using a stack to nest procedure calls and bit manipulation to operate the LCD module. A legible, appropriately commented source file will be appreciated and credited appropriately.

**EQU vs DEFW**

There is sometimes some confusion between the uses of the '`EQU`' and '`DEFW`' (and allied) directives; these have different purposes.

| `EQU` | `DEFW` |
|---|---|
| Syntax:<br>`label equ <expr>` | Syntax:<br>`{label} defw <expr>{,<expr>...}` |
| Defines a **compile time** symbol with the specified value. | Defines a (several) **word(s) in memory** with the value(s) set by the expression(s). |
| **Label** is set to **value** of expression. | **Label** (if present) set to the **address** of (the first) used location. |
| **No presence** in object code. | **Word(s) present** in the memory image. |
| Used to define **constants** etc. | Used to **reserve space** for and **initialise** variables, look-up tables etc. |

`EQU` provides a way to define a *label* to a chosen value (expression). Labels provide convenient aliases for values *when building* a code image; they have no presence in the finished code.

`DEFW` et alia provide a means of defining (one or more) data elements which will be loaded into the target's memory.

## Word Alignment

There is sometimes some confusion about which addresses can be used for storing different 'items'. This box describes why some restrictions exist and which apply to the RISC-V processor used in this lab.

### Restrictions

This example uses 32-bit words but the principle applies to any word size.

Computer memory is arranged in words to allow a machine to run quickly: if more bits can be moved from or to memory at the same time then fewer cycles are required. On the other hand, more parallel wires require more costly hardware.

In a (typical) 32-bit machine, memory will be 32 bits wide: each 32-bit word has an address. However, historically, it was convenient (and economic) to be able to use smaller variables (e.g. for characters) and the 8-bit byte became the standard. Architectures since have used word sizes of $8 \times 2^N$ bits – i.e. {8, 16, 32, 64}; the $2^N$ multiplier makes sense since multiplying and dividing by these quantities is trivial in binary whereas multiplying and dividing by (say) 3 or 5 is much harder.


Only bytes which are *stored* together can be moved in a single cycle. Thus a word fetched from memory should comprise bytes which are stored at an appropriate byte address multiple.

Some machines *will* allow the programmer to transfer arbitrary quantities at arbitrary addresses: they do this by using additional hardware to insert extra memory cycles. This is slow and inefficient though and is generally a Bad Idea, even if it is allowed.

### The RISC-V (RV32I)

RV32I is a 32-bit processor and (historically) insists on word alignment. Thus:

- **bytes** can be located at any address.
- 32-bit (4 byte) **words** can be located only at addresses which are **multiples of 4**.
  - note that **instructions** are words so this restriction applies.
- 16-bit **halfwords** can be located only at addresses which are multiples of 2, but are not really required in this module.

In the assembly language, using the directive `ALIGN` will move (if necessary) to the next multiple-of-four address.

The addresses used for words are (by convention) the address of the *least significant* contained byte: thus the bottom two *bits* should always be zero.

**Non-compliance will cause an alignment trap – not described until chapter 12.**

## Programming Tip

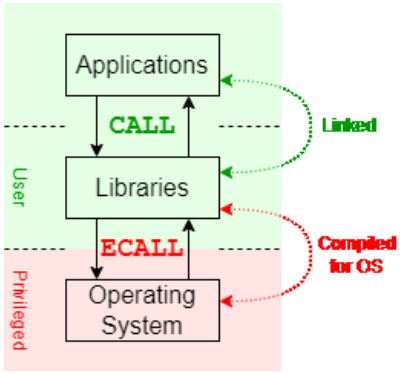
**Don't branch if you can avoid it.** Branch instructions do no actual *computing* and can disproportionately slow execution due to pipeline flushes. If you can find a formula instead of an '`if`' your code may be faster. This principle extends into high-level source code.

Example: word-aligning a (32-bit) pointer (a.k.a. the `ALIGN` directive).

```
pointer := (pointer + 3) & 0xFFFF_FFFC;
```

# 11 | Code Organisation

## 11.1 What belongs where?




To some extent, how you organise your code is up to you. However there are certain rules which should be observed and others which will help you (and others) in the long run. This section briefly describes three different places where code can be located and outlines what is appropriate in each place.

**Operating system** functions are run in a privileged mode. These will usually be invoked via a system call, but could also be invoked by another exception such as an interrupt (q.v.). The operating system deals with the hardware and all direct hardware reads and writes should be contained within. It also provides an **abstraction layer** which is specific to the particular machine at one 'side' but has a generic interface for the user. This allows the same user code to work on many different machines.

**Libraries** contain commonly used functions which may be called by a number of different programmes. Keeping these functions in a library saves writing them more than once. To be generic, library functions are often quite 'plain' and often require several parameters. They run in user mode but may (and often do) call operating system routines.

**User code** is the 'real' application programme which contains all the 'custom' code for the particular job. It runs in user mode and calls both libraries and the operating system as necessary.

**Library contents**

What sort of things should go in a library? This should be user mode code — i.e. no modification of hardware except via the defined system interfaces — which is used frequently. An example here would be printing a string of ASCII characters: printing an individual character to the LCD requires hardware manipulation and is therefore an operating system task; *iterating along* a string can be done in user mode and, as it is often required, maybe should be a library function.

Printing a number in ASCII is another example. Here you could consider whether another parameter to specify the radix (e.g. decimal, hexadecimal, etc.) would be worthwhile.

For some other examples try typing "`man string`" (for example) at a shell prompt. If the manual is installed this will list some of the standard C language libraries. You don't have to interpret all of them here!

### 11.1.1 Subsection on interfaces?

**Building a library**

First consider if a function is likely to be used in several different programmes. If it is then it is a good candidate for being placed in a library.

Next think about how that function may be applied in the future. Are your first ideas more generally applicable or would the addition of an extra parameter (even if ignored for now) make it more useful later. Remember, once in use several programmes may rely on the same function so the interface should be fixed.

Define the interface. What goes in and what comes out? What registers (or memory locations) are used for these? Document this!

Put the function in a separate source file, with other, similar functions.

Include the library source file in your programme.

```
INCLUDE <filename>      ; "GET" is synonymous
```

This will instantiate the code *at that position*. It may therefore be best to 'include' libraries at the end of your main source file. Call the function as normal.

[Note that you are including the library functions as if you had copied them in; therefore labels should all have unique names (i.e. *scope* is global.]

## 11.2 What belongs *when*?

There should also be a clear distinction as what *can* be done *when*. Some calculations are *dynamic*: they rely on run-time information. For example the *particular* entry in a jump table is only loaded after you know which entry you want ... this time.

However, other calculations may be done *statically*, i.e. at compilation/assembly time. For example, if you want to know how many entries there are in a table (of, say, 4 byte words) this will be a fixed quantity in the object code and the assembler (or compiler) can work it out for you.

e.g. "`(table_end - table_start) / 4`"

Although you may sometimes edit the code, you must always reassemble (recompile) it before use, when this number can be calculated; it never changes in between builds.

Clearly it is more efficient to precalculate values (once, and 'off line') whenever this is possible to avoid performing the same calculation repeatedly at run time. Always consider what might be possible at compile time and try and ensure that it is done then.

**Verification hints**

Inset or section?

It is recommended that you verify code as you develop it (and retain test data for regression resting). This should include testing of things you might *not* normally expect, such as error cases. Deliberately introduce errors (temporarily!) to check that code intended to detect them also does its job.

A particular case to watch for (verify!) is the '**off-by-one** error' where a boundary is in slightly the wrong place. A couple of examples could be writing:

- an array index from `1 - N` rather than `0 - (N-1)`
- a comparison being $>$ rather than $\geq$

Experience suggests this will become pertinent to some of the following exercises!

**Example Programme Header**

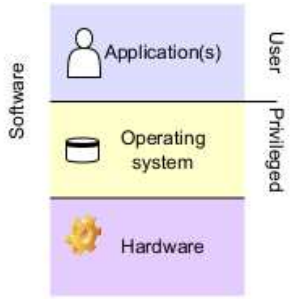
It may not matter for small programmes, but as projects grow it saves time if you can identify what is in each file and when it was last modified. This is an example of the sort of information which is useful.

```
;-------------------------------------------------------
;               Traffic lights programme
;               J. Garside
;               Version 1.0
;               1st January 2003
;
; This programme emulates a set of traffic lights
; using the on-board LEDs.
;
;
; Last modified: 2/1/03 (JDG)
;
; Known bugs: None
;
;-------------------------------------------------------
```

As complexity increases it is a good idea to document each procedure etc. in a similar way. This is especially useful when collaborating on projects where, for example, you can check if someone has changed a routine since you last used it.

# 12 | RISC-V Operating Modes

## 12.1 Privilege modes




A modern microprocessor will typically have a number of *privilege modes* which allow *protection* of parts of the machine from malicious, mischievous or malfunctioning software. A common use is to have a privileged Operating System (OS), which can manipulate the hardware, and unprivileged applications, which cannot. All the software (traditionally, at least) runs on the same processor so some means of determining the current priority is needed. To this end the processor will be executing in a given **mode**, which the hardware can monitor to detect unauthorised operations.

The primary use of privilege modes is to permit or forbid access to areas of the address space(s); for example, memory-mapped I/O devices will typically be protected from *direct* manipulation by applications, a.k.a. 'user' access. Other privileged operations include access to OS code and data spaces and control of system functions such as interrupts; it should be obvious that, for security, unprivileged code cannot increase its own privilege except by limited and well-protected means.

Applications should always be treated with suspicion and run unprivileged. When they need to perform some system operation, such as I/O, they can gain privilege by making a **system call**. This not only increases the privilege but jumps to another address – which is safely set up in a memory area where the *handler code* resides. This area is inaccessible (directly) to applications, which prevents the application planting its own code there. The privileged service routine is *trusted* to do nothing dangerous; it's part of the operating system. When the service is complete the code *returns* to the caller, restoring the previous privilege at the same time.

### 12.1.1 RISC-V privilege modes

The RISC-V specification encompasses a wide range of implementations and not all possible modes need be present. A very small implementation may only have a single mode (a view you have seen before), but here we need to introduce more detail.

There are currently three basic modes (plus variations) specified: in order of increasing precedence these are called '**User**' mode, '**Supervisor**' mode and '**Machine**' mode. In a workstation these might be used for running applications, operating system(s) and a *hypervisor*, respectively. This module need not go that far and only needs the User (**U**) and Machine (**M**) modes.

At reset the processor is put into in machine mode; this makes sense since it needs privileges to initialise the hardware etc. When it runs an application it will (typically) do this in user mode.

## 12.2 Traps

Besides **reset**, there are numerous ways to enter the operating system although most of these result from some hardware intervention. These are referred to as "**traps**". An example would be trying to execute a floating point instruction on a processor with no floating point unit: this results in an *illegal instruction* trap

and the operating system can intervene to recover or terminate that application. One alternative here is to *emulate* the unknown instruction in software ... but we digress.

A trap is caused by an '**exception**': typically (but not always) when something unexpected happens as a programme executes. RISC-V divides its traps into 'synchronous' traps – which are caused by something happening at a particular place during execution – and 'asynchronous' traps – which are, basically, interrupts. We'll deal with interrupts later (chapter 17).

Examples of synchronous traps include ingesting an instruction code which is discovered to be 'illegal', attempting to read or write a memory area which is not allowed (with the current privileges), *misaligning* a memory operation (e.g. a 32-bit word at an odd-numbered address) etc; these are all hardware-trapped variations of otherwise normal execution.

The 'deliberate' way for an application to invoke an OS service is a '**system call**': this is an operation which can be embedded in the code in the same way as an internal call and will run a service and (usually) return to the following instruction. On the RISC-V this instruction is known as an 'Environment Call' and has the assembly language mnemonic '`ECALL`'. The instruction has no arguments – other information needs to be passed (for example in registers) to specify the particular service required.

The complementary, *return*, operation has the mnemonic '`MRET`'; it is a *privileged* operation so trying to execute this in a user application will result in an illegal instruction instead!

To describe the behaviour of these operations first requires delving into more of the processor's details.

## 12.3 Control & Status Registers (CSRs)

Before describing the `ECALL` action it is necessary to introduce more features of the RISC-V architecture. In addition to the RISC-V's 32 registers (plus `PC`) and its 1 Gword (4 GiB) memory address space there is another space containing the CSRs (fig. 12.1). This is a 4 Kword space (partially) filled with a variety of registers used to configure and report on the processor's operation. These include information such as the processor's manufacturer. Some of the CSRs are available to the user but, as might be anticipated, many are themselves privileged.

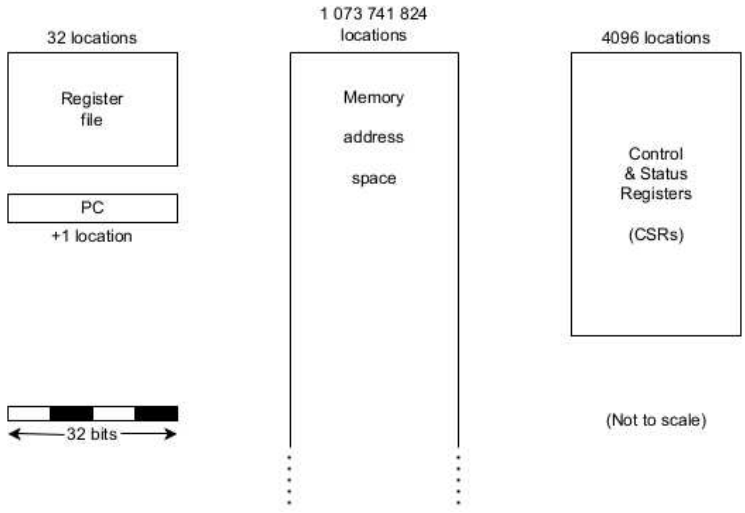



Figure 12.1: RISC-V address spaces

The RISC-V documentation [REF] has a map of this space although this is not (yet) full. Not all the defined CSRs need to be present in any given implementation and there are spaces for custom design. The CSR *space* is also subdivided for access protection according to privilege mode and some areas are read-only. Most of these details do not concern these exercises and we will keep to a small subset of **M**achine mode CSRs: table 12.1 shows the CSRs of immediate concern by name (the assembler will recognise these abbreviations, in upper-case) complete with their addresses.

| Register | Abbreviation | Name | Purpose |
|---|---|---|---|
| 300 | MSTATUS | Machine status | Collected machine status bits. |
| 305 | MTVEC | Machine Trap Vector | Address jumped to on trap entry. |
| 340 | MSCRATCH | Machine trap handler Scratch | Free for software use. |
| 341 | MEPC | Machine Exception PC | Address of instruction which 'trapped'. |
| 342 | MCAUSE | Machine trap cause | Reason for trap (table 12.2). |

Table 12.1: Immediately relevant CSRs

These registers are involved in all M-mode entries or '**traps**'. Traps encompass both '*Exceptions*' – ECALL being one of these – and '*Interrupts*'. First let's look at the trap *behaviour*.

On any processor which supports them, a system call must change both the privilege mode and jump to a service routine in privilege-protected memory. As it has to *return* it must first **save the current PC and privilege mode** ... somewhere. The RISC-V, being a RISC processor (there's a clue in its name!) uses a simple approach by copying these into convenient registers (fig. 12.2). The PC is saved in MEPC; the Previous Privilege mode – which is a 2-bit code – is saved in a field of the MSTATUS register. The processor can than **rewrite the current privilege and jump** to a predefined handler. Note that this is **atomic** i.e. all the changes are simultaneous.

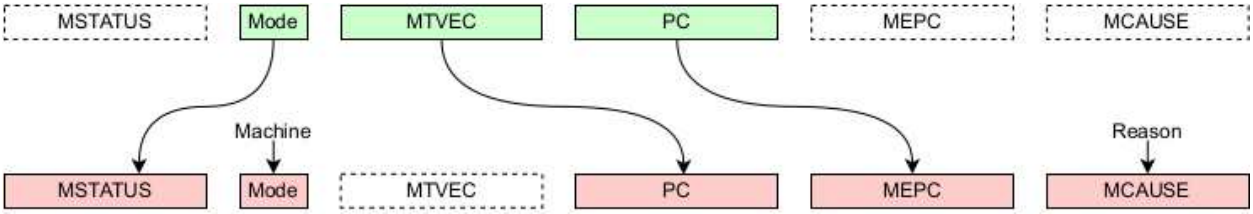



Figure 12.2: Data movements during a trap

Note: in all traps, including ECALL, MEPC holds the address of the 'problem' instruction, thus, unlike a 'normal' call, it is not *directly* the return address,

| Number | Reason |
|---|---|
| 0 | Instruction address misaligned |
| 1 | Instruction access fault |
| 2 | Illegal instruction |
| 3 | Breakpoint |
| 4 | Load address misaligned |
| 5 | Load access fault |
| 6 | Store address misaligned |
| 7 | Store access fault |
| 8 | Environment call from U-mode |
| 9 | Environment call from S-mode |
| 10 | *Reserved* |
| 11 | Environment call from M-mode |
| 12 | Instruction page fault |
| 13 | Load page fault |
| 14 | Reserved for future standard use |
| 15 | Store page fault |

Table 12.2: Exception causes

The new mode will be Machine mode: the new PC is supplied by MTVEC which should have been **initialised** with the address of the exception handler.

The exception *type* is written into MCAUSE; the current set of *synchronous* exception causes are listed in table 12.2. Initially only cause 8 should need to be handled but it is quite likely that some error(s) might cause one of the others.

MSTATUS (fig. 12.3) is a collection of bits, most of which are irrelevant here; indeed some are not implemented because they are not needed. The bits which *are* needed are introduced later, starting in subsection 12.3.2.

The function of MSCRATCH is entirely defined by software: however, as will be illustrated in subsection 12.3.3, it can be used very conveniently in the exception handler entry and exit.

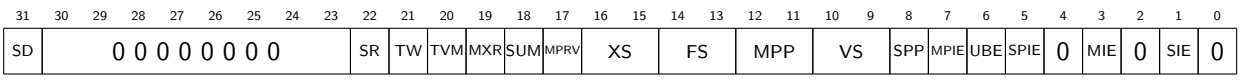



Figure 12.3: Status register bit fields

### 12.3.1 Access to CSRs

Perusing the full RISC-V documentation will reveal the full set of CSR instructions but they are not all needed just now; one operation and its convenient pseudoinstruction variants will (almost[1]) suffice. There are some more details in an inset on page 64 which you can read or skip, according to taste.

The relevant mnemonics/formats are:

```
CSRR    <Rd>, <CSR>          ; CSR read to register
CSRW    <CSR>, <Rs>          ; CSR write from register
CSRRW   <Rd>, <CSR>, <Rs>    ; CSR register swap
```

The last (which is the true instruction – the others are pseudoinstruction variants) performs a simultaneous read and write and the source and destination register(s) may be the same. It can therefore be used to *swap* a register with a CSR if desired.

### 12.3.2 Changing privilege mode

The processor privilege is, deliberately, difficult to get at; indeed the designers do not allow it to be read directly[2]. Mode changes to increase privilege are caused by trap operations. There are two *software* trap operations – **ECALL** and **EBREAK** – which are intended for 'everyday' and debugging use, respectively; in addition there are several hardware traps (table 12.2).

To lower the privilege mode – usually to *return* from a trap – the corresponding return instruction will restore the previous context (fig. 12.4). From **m**achine mode, the appropriate instruction is **MRET**.

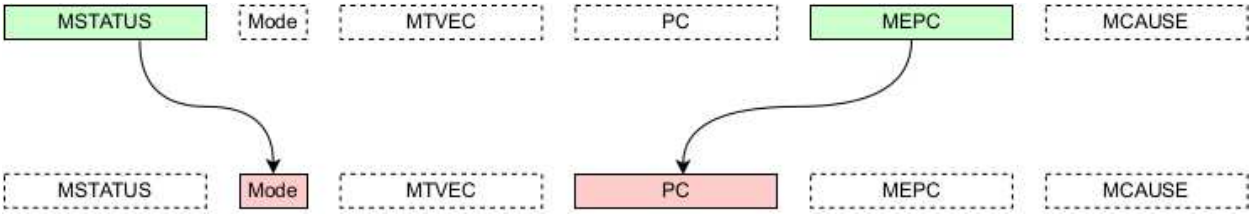



Figure 12.4: Data movements during a trap return (MRET)

#### Getting to User mode

This begs the question "How to reach user mode the first time?". The mechanism is the same – MRET – but, since there has not yet been a user-ECALL its effects must be 'forged'. Referring to figure 12.4 the necessary values are the address of the user code in MEPC and the 'previous' priority mode, which is a field of MSTATUS.

This is codified in the next subsection under 'Initialisation'.

### 12.3.3 Trap handler structure

Once we start to reduce the processor privilege we significantly increase the chance of (inadvertently) causing a trap. Thus it is important to set up code to catch these, if only to aid debugging. The trap handler(s) will become increasingly important in subsequent exercises.

These are *examples* of code here: they are not mandated by the processor ISA.

[1] See also CSRC on p. 64.
[2] This can make *virtualisation* easier.

### Initialisation

At initialisation, set `MSCRATCH` to the Machine stack pointer and `MTVEC` to the address (in privileged space) where the machine mode trap is to be entered (e.g. '`mhandler`'): example code might look like this:

```
...
li      t0, 0x0000_1800     ; Load MPP mask - bits 12 & 11
csrc    MSTATUS, t0         ; Clear MPP bits in status
la      t0, mhandler        ; Point at trap handler code start
csrw    MTVEC, t0           ; Save address in system CSR
csrw    MSCRATCH, sp        ; Copy 'machine' SP for use in handler
la      sp, user_stack      ; Change SP to user space
la      ra, user_code       ; Point at user code start
csrw    MEPC, ra            ; Save as 'return address'
mret                        ; 'Return' to programme start
```

This sets:

- the 'saved' privilege mode (MPP) in `MSTATUS` to '`00`' (user mode).
  (The `CSRC` instruction is described in the inset on page 64.)
- the trap vector CSR to point to the trap handler code.
- the trap 'scratch' register to point to the current (system) stack top.
- the (forged) 'previous trap' to 'have been' at the desired start address.

and then 'returns' in the appropriate (user) mode to the desired place (`MEPC`) using an atomic branch which moves these values to the current privilege and `PC` registers.

### Handler code

A trap handler (also in privileged address space) is *similar* in most respects to any other call. There are some minor variations however which should be considered.

- Some exception traps will not return. An example would be a *segmentation fault* which is a non-recoverable fault as far as an application is concerned.
  We will not be addressing these directly in this module.
- In *most* cases all the user register state (etc.) must be preserved. Most traps are **exceptions** so happen *unexpectedly* – but often are recoverable from as long as no user information is lost.
  We will first deal with `ECALL` which is *not* unexpected but it should be borne in mind that retailing user state is usually desirable and we will be revisiting the code later.
  `ECALL` is an exceptional[3] trap in that sometimes it *is* used to return function values in a register.

This is not following the ABI – but we are just dealing with **A**pplications here!

Probably the first thing to do is to save some working registers. An `ECALL` may arbitrarily change some registers – and sometimes will deliberately return values – since these may be anticipated in the programme flow; later, when it comes to interrupts, the service routines *must* preserve *all* the processor state[4]. However some registers will be needed during the call so some of the user's values must first be saved. They can be *stacked* but best to use the handler's own stack. This is where `MSCRATCH` can be useful.

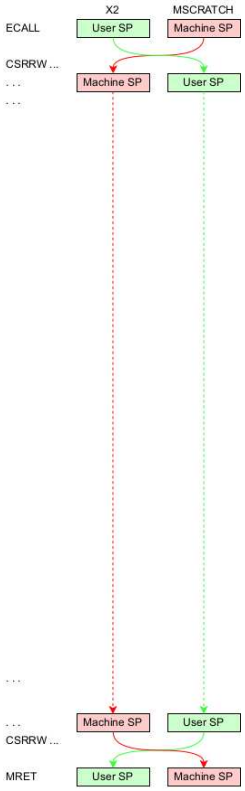
Three registers are saved here – an illustrative, arbitrary choice – including the (default) link register (`ra/x1`).

*** This needs reconciling with the description in section 14.2. ***

[3] Sorry about the pun!
[4] This becomes **really important** when we get to **interrupts**!

```
mhandler    csrrw   sp, MSCRATCH, sp    ; Save User SP, get Machine SP
            subi    sp, sp, 12          ; Push working registers
            sw      s1, 8[sp]           ;
            sw      s0, 4[sp]           ;
            sw      ra, [sp]            ; Handy for later 'calls'

            csrr    t0, MCAUSE          ; Read why we came here
            li      t1, 8               ; User ECALL cause
            beq     t0, x9, ecall_handler
            ...
```




The exit sequence needs to reverse this, something like:

```
            ...
            csrrw   t0, MEPC, t0        ; Find the trapping instruction
            addi    t0, t0, 4           ; Correct to a return address
            csrrw   t0, MEPC, t0        ; Swap back in

            lw      ra, [sp]            ; Pop working registers
            lw      s0, 4[sp]           ;
            lw      s1, 8[sp]           ;
            addi    sp, sp, 12          ;
            csrrw   sp, MSCRATCH, sp    ; Save Machine SP, get User SP
            mret                        ; Return
```

**Note that `MEPC` has saved the address of the trap – in the case of `ECALL` this is the address of that instruction; thus when returning from an `ECALL` the saved address must be moved to point to the *next* instruction, hence adding 4. *This is not general to all trap returns.***

A more 'advanced' approach to the exception dispatcher is to use a jump table (section 8.3) with the cause which provides a 'case' (or 'switch') structure; this may be useful later. A possible coding might look like this:

```
            csrr    t0, MCAUSE          ; Read why we came here
            andi    t0, t0, 0xF         ; Cautious - guarantees in range
            la      t1, trap_table      ; Pointer to table
            slli    t0, t0, 2           ; Multiply cause to word offset
            add     t1, t0, t1          ; Index into table
            lw      t1, [t1]            ; Get handler address
            jalr    t1                  ; Call specific handler (RA is implicit)
            ...                         ; Can ret(urn) to here


trap_table  defw    trap_handler_0      ; Addresses of handler code
            defw    trap_handler_1      ;
            defw    trap_handler_2      ;
            defw    trap_handler_3      ;
            ...
```

There will be more trap dispatching later: if curious now you could glance ahead to section 17.6.

## 12.4 Alignment traps

If you have violated the memory alignment (p. 54) you may already have experienced a trap: RISC-V performs a trap entry if a word (or halfword) access is improperly aligned. Our lab. set-up initialises `MTVEC` to `0000_0000` so this will look very similar to a reset. You now have the means to isolate these faults — there are three types {instruction address, load address, store address} — although they indicate coding mistakes and should be fixed.

**Trap returns**

When it traps, RISC-V saves the PC at the point of the trap. This is the 'wrong' place for an `ECALL` to *return* to since it has been executed; on the other hand for traps such as page faults it is the appropriate return address so that the faulting instruction can be retried. The details need to be considered on a case-by-case basis.

The interrupt mechanism obtains its return `PC` by 'hijacking' an existing instruction which will therefore not be executed: thus this is the correct return address since the hijacked instruction needs to be 'retried'.

**CSR access instructions**

There are, basically, three types of CSR access instructions; each has an option to use an immediate source field but these are not generally particularly useful since the immediate field only reaches the lower five bits. These instructions are supported by *pseudoinstructions* – convenient assembly language short-form mnemonics.

As documented in the main text, `CSRRW` simultaneously reads and writes the specified CSR.

- If the destination register is specified as `x0` then the read is not performed; there is a shortened mnemonic for this: `CSRW`.

The other CSR instructions are `CSRRS` and `CSRRC`. These read the specified CSR and **S**et and **C**lear (respectively) the bits where the source has a logic '1'.

The ability to alter some bits in a register whilst preserving others is sometimes important in I/O registers, as was mentioned in subsection 7.1.1; it allows *atomic* operations. Altering bits with a read-modify-write software sequence can fail if bits in the register can change between the load and store operations, which can happen with some peripheral registers which can also be changed independently by their hardware operation.

- There are shortened mnemonics if the read value is not required: `CSRS` and `CSRC`.

To complete the RISC-V story, `CSRRS` or `CSRRC` can be used to read a CSR without the write operation.

- If the source register of `CSRS`/`CSRC` is specified as `x0` then the write is *not performed* although the read takes place; there is a shortened mnemonic for this: `CSRR`.

This (obviously) changes no bits but actually avoids the write altogether which may be needed in case the act of clearing-no-bits of the CSR causes side effects.

**Selective summary**

| Mnemonic | Function | Implementation |
|---|---|---|
| `CSRRW Rd, CSR, Rs` | Read and Write CSR | `CSRRW Rd, CSR, Rs` |
| `CSRW CSR, Rs` | Write CSR | `CSRRW x0, CSR, Rs` |
| `CSRR Rd, CSR` | Read CSR | `CSRRS Rd, CSR, x0` |
| `CSRS CSR, Rs` | Set CSR bits | `CSRRS x0, CSR, Rs` |
| `CSRC CSR, Rs` | Clear CSR bits | `CSRRC x0, CSR, Rs` |

# 13 | Memory protection

*It may be valuable to refer to your notes on Operating Systems regarding memory space.*

A *simple* microprocessor system may allow unrestricted access to all locations in its memory space(s). However to add security to a computer it is desirable to offer protection to certain parts of the address space. For example, if there are two users of a machine, one should not be able to sabotage the code of the other – and neither should be able to crash the whole system. In particular, no **user** should be able to access the I/O space directly as this could cause all sorts of undesirable effects. In a desktop computer the I/O devices will always be protected by an Operating System (OS) — trusted code which the user cannot hack (either accidentally or deliberately) — which will have a device driver for the particular hardware.

Most contemporary processors include some hardware memory protection which checks each attempted access and may disallow some if they are potential security hazards. The failure of such a cycle is usually called an '**abort**' and evidences as a **trap**.

Aspects which may be checked include:

- **source** of request: instruction fetch or data transfer.
- transfer **direction**: read or write.
- current **privilege level** of processor: operating system or application.

In the general case permissions will be programmable (by the OS) as part of the *memory management*; in this laboratory they have been fixed to avoid this extra complication.

## 13.1 Aborts

This section is here as a warning that you may (inadvertently!) encounter access faults once you start changing the privilege mode so take care that your code (and variables) are in the appropriate regions.

In this lab. some of the address space (the lowest $\frac{1}{4}$ MiB) is privilege-protected and not accessible in user mode: attempts will result in a **trap**. This area covers 16 KiB of RAM (which should be reserved for 'operating system' functions) and the memory-mapped I/O devices. The most likely circumstance in which you may encounter an abort in this lab. are:

- the `PC` jumped to a protected RAM area while the processor is in *user* mode ⇒ **instruction access fault**.
- the code crashed and `PC` runs 'off the end' of the RAM ⇒ **instruction access fault**.
- an attempt to read or write an I/O port while in user mode ⇒ **data access fault**.
- trying to read or write protected RAM whilst the processor is in *user* mode ⇒ **data access fault**.
- trying to read/write a *word* at an address which is not a multiple of 4 ⇒ **load/store misalignment**.

Table 12.2 (p. 60) shows the codes for these (and other) traps.

The $\frac{1}{4}$ MiB of **User RAM starts at `0004_0000`** (see subsection 3.5.1). You can alter the addresses used by the assembler with the `ORG` directive. If uncertain you can look directly at the `.kmd` output file: it's plain ASCII so easy to read.

### 13.1.1 What happens if I try and access a protected area (whilst not allowed)?

Firstly the access cannot be allowed so the memory (or I/O) state is not changed. Secondly something 'wrong' is happening so the processor **traps** (enters an exception) to what is assumed to be safe, operating system code.

The trap causes basically the same action as taken by a deliberate `ECALL` (system call) – the subject of chapter 14 – so they both need initial handling in the same way. RISC-V identifies several **trap causes** which can be seen in table 12.2 on page 60. There are three categories of trap on each memory access:

**Misaligned** is where the address is inappropriate for the data size, such as an instruction fetch or word load from an 'odd numbered' address. (See inset on page 54.)

**Access fault** is where the transfer is not permitted, such as a user application trying to use private OS address space. This is, effectively, the classic "**segmentation error**".

**Page fault** is where the transfer is permitted *in principle* but cannot take place at this time. We don't have paging on our lab. system.

Each of these is classified separately according to the memory operation attempted: instruction fetch, data load or data store, so $3 \times 3 = 9$ different 'cause' codes.

In all cases the RISC-V performs a trap and the handler must then perform the appropriate action (see subsection 12.3.3). `MEPC` will point to the guilty instruction; another CSR (`MTVAL`) will contain the failing address (which is different from the instruction address for *data* transfers).

### 13.1.2 What might I see and what should I do about it?

Look for the processor entering machine mode unexpectedly. This suggests there is a problem.

'Contain' all traps unless and until there is an appropriate handler: this might simply entail stopping (e.g. "`J .`") and using the debugger to diagnose the problem. There is considerable information saved in various CSRs.

- `MEPC` will reveal the faulting instruction.
- `MCAUSE` will identify which trap was taken.
- `MTVAL` will contain an argument for some trap causes.

Of the traps listed in subsection 13.1.1 only the page fault is *recoverable*[1]: the others are errors. Find what went wrong and make sure you only access I/O space (and 'system memory' from a privileged mode (i.e. use an `ECALL` as described in exercise 4).

[1] . . . and we don't generate these here.

**RV32 programming puzzle**

Determine the magnitude of `a0` . . . i.e. `ABS(a0)` *without using branch instructions.* (Branches often take longer than they might appear to, because of pipeline disruption.) Other registers may be changed. (This is quite difficult; you might need to look it up.)

Target: 3 instructions

**Memory aborts in an OS**

There can be different reasons for encountering a memory abort in an OS. Some are due to errors or attempted security breaches; when these are encountered the OS will identify them and kill the offending process. You have very likely seen these as a "**segmentation fault**" or similar error if you have ever written any C code.

Aborts do not always indicate faults. In a paged, **virtual memory** system they may indicate a **page fault**, where access is legal but the required memory is not currently loaded. These trigger the paging action – usually swapping to a different application whilst this proceeds – but eventually *return* to the code to re-run the instruction which should now succeed. (Note, unlike an `ECALL` etc. this should return to the *'failing'* instruction so it can be retried; `ECALL` returns to the instruction *following* the call.

The **RISC-V** memory 'fault' model can distinguish between memory aborts and page faults as different 'causes'.

Fault determination can be more selective than the simple system implemented here: it may distinguish load from store operations so that an area is 'read only' or instruction fetches from data loads so code can be run but not examined directly.

Other exploitation is sometimes used. For example, in a paged system the memory fetched from disk can be marked as 'read only' even though the page is legitimately writeable. If a write is attempted it will abort; the permission is checked in software and the page made writeable before proceeding. This only need occur *once* on that page. When the page is subsequently picked for eviction the permission an be checked to see if it is '**dirty**' – i.e. at least one write has taken place. A 'clean' page need not be written back.

**Illegal instructions**

An 'illegal' instruction is an opcode which the instruction decoder does not recognise. There are many of these among the 4,294,967,296 possible 32-bit RISC-V instruction codes: significant *areas* are used for various extensions. In particular, all the basic 'legal' opcodes have the two LSBs set to '`11`'; this makes `0000_0000` an illegal code.

> RISC-V instruction codes which do not have the two LSBs set to '`11` are used as shortened, 16-bit codes for a subset of common instructions if the 'C' extension has been included in the processor. Shortened codes reduce the memory requirement and, potentially, improve code-fetch bandwidth.

If an illegal opcode is ingested it will cause an illegal instruction **trap** (cause #2). This can be used for various purposes: for example it could be an *emulator trap* where an unrecognised code (say, a floating point instruction on a processor with no floating point unit) may be identified and modelled in software. Although much slower than using hardware this allows the same object code to run on a range of systems.

> It may be possible (on *some* RISC-Vs) to enable/disable extensions *dynamically* (i.e. at run time) thus making some codes *optionally* illegal.
> Challenge: can you think of a way to exploit this? [Difficult!]

# 14 | Exercise 4: System Calls

## Objectives

- Vary operating privilege
- Introduce system calls
- Start to handle **traps**

System calls are another type of trap: their function is to request a privileged service from the machine and they (almost) always return control to the application.

## 14.1 System Initialisation

When a system is switched on a large amount of its state is **undefined**. ROM will hold its contents (of course) but modern RAM relies on a power supply to retain data so it will be in an unknown state (unless battery-backed). More relevantly the processor's registers (including the `PC`) will be undefined which means the processor's behaviour cannot be predicted.

This would clearly be unacceptable! Therefore at power-on (and possibly at other times) a subset of the system state is **reset** to predefined values. It is not normal to attempt to define all the state (for example the RAM contents) as this would be too expensive, but registers such as the `PC` are defined, so that the processor will execute a known piece of code. If more initialisation is required (it usually is) then the software can perform this function.

### 14.1.1 RISC-V Initialisation

Following reset the RISC-V's register state is undefined except for:

- `PC` which is set to a predefined value; this may vary according to implementation but here has been defined as `0000_0000`.
- The **privilege mode**, which is 'Machine'.
- Various CSR fields, notably the **interrupt enables** in MSTATUS which are cleared (interrupts disabled).

If some of these terms are unfamiliar, don't worry just yet.

There is more to do so it is usual to run some initialisation code before control is passed to a user programme. This can do numerous things but here we are only concerned with a small set of operations:

- Initialising exception vectors.
  - Various trap routines — such as interrupt handlers — may need installing. At present these can all be downloaded with the programme.
    There is a single entry point from where the routines must be branched to, according to the `MCAUSE` CSR.
  - The CSR '`MTVEC` must be loaded with the entry address of the trap handler.
- Initialising stack pointers.
  - At some time it is likely that the software will require a stack. An area of memory must be allocated for this purpose and the stack pointer must be set to one end of this area.
    In practice it is normal to have a separate stack for user and supervisor code. Any stack pointer(s) which may be required in future should be set up now.
- Initialising any peripherals required.
- Entering user mode.
  - Before a 'user' programme is executed user mode should be entered.
    This should be the last thing on the agenda because user mode code cannot switch back into a privileged mode except through a trap.

When initialisation is complete it is sensible (for protection) to reduce the privilege to run the application. A description – and an example code fragment – was given in subsection 12.3.2 (p. 61).

### 14.1.2 Address space segmentation

It's normal to divide the memory map according to function. In the lab. system the (crude) protection system is hard-wired: the I/O peripherals are protected and the RAM is divided into a protected area – intended for initialisation code and trap handlers – and an unprotected area for running applications. Code should now be *segmented* appropriately to use these spaces with handlers limited to code which deals directly with hardware (or a few, private OS variables).

'Manual' entry to privileged code is achieved with a **system call** (see chapter 12): RISC-V uses the term 'environment call' (`ECALL` for short)[1] so we will adopt this name within this module. `ECALL` causes a **trap**, as described in section 12.2.

> You should have used some simple **ECALL services** — such as printing a character — before (in COMP15111). Note that these were provided by the emulator on your workstation and **are not available** here. **You must now write your own.**

Before making an `ECALL` the handler code must be set up and `MTVEC` initialised. The trap behaviour and some suggested code templates may be found in chapter 12.

There are many trap *causes* and it is likely that you won't immediately want to write code for all of them. However it is possible – even *likely* – at this point that some unexpected traps – particularly aborts (section 13.1) – may occur due to errors. For this exercise you will need to separate at least the `ECALL` 'cause' (which is number 8 in the `MCAUSE` CSR); there is some code which will do this suggested in chapter 12 (specifically p. 63). There is an expanded illustration of the process – including some aspects which are not *yet* relevant – later in section 17.6 (fig. 17.4).

What behaviour should be observed if an unexpected exception occurs? *Ideally* there should be an attempt at error recovery, or at least reporting to the operating system. At this point however this would be over sophisticated. Some possible behaviours are:

- Restart the code (i.e. same as Reset).
- Return, ignoring the exception.
- Halt.

These can be varied for the different causes. If you want to try and return from an exception (such as a spurious interrupt) discuss your solution first; there are some subtleties which are not immediately apparent.

[1] These are also known, in other systems, as 'traps', 'restarts', 'supervisor calls', 'software interrupts', etc. according to taste.

### 14.1.3 Context and Context Switching

The **context** of a programme is the environment in which the code runs. One component of this environment is the register contents; clearly when switching from programme to programme each needs its own set of values. (Another component is the memory map, but we are not dealing with memory management here.)

In a multi-tasking system it is usual to provide one stack for every process; during **context switching** the stack pointer will be moved from one process' stack to another. Whilst we do not (yet) want to do this, the 'obvious' context switching — between user programmes — is not the only context change which the system will undergo. For example interrupts (coming in a later exercise) will normally have their own context.

More germane here is the fact that the RISC-V has (at least, partially) separate contexts for user programmes and the operating system. Thus there will typically be a user (application) stack in the application memory and a system stack in a more 'private' (protected) area. These areas of memory need to be defined and reserved. The stack pointers also need to be set up, although (by convention) the RISC-V only uses `SP`/`x2` as its normal stack pointer. A convenient place for the OS stack pointer is the `MSCRATCH` CSR, from where it can be swapped in and out at the beginning and end of the trap service routine.

```
CSRRW      sp, MSCRATCH, sp
```

### 14.1.4 Peripheral initialisation

In this exercise there is only one peripheral which needs configuring before the user code can be started, namely the LCD. This should be cleared and the interface signals left in a defined state. This is done[2] by a hardware reset (such as power up) but you may wish to reinitialise (e.g. clear) the display each time your programme starts.

In general there may be a number of different devices which should be configured at this time. For example if interrupts are to be used (which will generally be the case) all the devices which could generate an interrupt must be initialised before interrupts on the processor can be enabled. More on this later.

### 14.1.5 Starting the user programme

When initialisation is complete the processor can be switched to user mode and dispatched to an application programme. Herein is a problem.

- If the OS – running in protected memory – simply changed the privilege the next instruction fetch would trap because the fetch would not be legal.
- If the OS jumped to user space a malign application could retain OS privileges and gain control of the whole system.

Thus the changes must happen *atomically* – in a single operation. This is described in subsection 12.3.3 under 'Initialisation' (p. 62).

As a final detail, the application process (or, in a multiprocessing system, *each* application process) will need its own stack . . . in user space. You may make this the responsibility of the application as it starts or you could choose to let the 'Operating System' set this before the branch: either way this involves setting up `SP` to point to the appropriate, reserved stack area (in user memory, this time). The choice of quite *when* to do this is left open as it is a system design issue.

[2] Actually it isn't in the Verilog module itself which has no 'Reset' input; we've done it for you to save fiddling about. The programming set-up is a bit messy since the LCD module powers on in an ill-defined state.

## 14.2 ECALL Service Routines

So (assuming you set up `MTVEC` correctly) you've taken a trap (of some kind). The first thing is to determine the trap type ('cause'). For this you will need to use some registers *but* in many cases you must not (irreversibly) change any user register state. The usual solution is to push some registers; for this we should use the 'machine' stack. Earlier on (subsection 14.1.5) we initialised `MSCRATCH` with this value so we can now swap it in, neatly preserving the user SP for later[3].

There is also a figure later (fig. 17.4, p. 86) showing this process combined with the asynchronous trap (interrupt) disambiguation.

### 14.2.1 Trap entry (repetition??)

The following assumes that the `ECALL` service number is already in `A7`. Read this code carefully and **make sure you understand the various steps**; it is not trivial on first reading!

```
trap_entry      csrrw     sp, MSCRATCH, sp   ; Swap in OS SP
                subi      sp, 8              ; Push working registers
                sw        t1, 4[sp]          ;
                sw        t0, [sp]           ;
                csrr      t0, MCAUSE         ; Find why we're here
                li        t1, 8              ; User mode ECALL 'cause' code
                beq       t0, t1, ECALL_code ;
                ...
                ...

ECALL_code      li        t0, ecall_max      ; Check argument is legitimate
                bgeu      a7, t0, ecall_x    ; Out of range default
                la        t0, ecall_jump     ; Point at table
                slli      t1, a7, 2          ; Calculate index (in words)
                add       t0, t0, t1         ;
                lw        t0, [t0]           ; Load address of service routine
                jr        t0                 ;  and jump to it.

ecall_jump      defw      ecall_0            ; Jump table of (code) pointers
                defw      ecall_1            ;
                ...
```

### 14.2.2 ECALL Dispatcher

It will soon become apparent that more than one different OS call is required. The RISC-V has only a single `ECALL` instruction which must be used to perform all its system functions. It is therefore necessary for the called routine to determine which service is required and dispatch to the appropriate handler.

This code is not completely optimised although for the `ECALL_code` it does use a **jump table** (see page 44). A predetermined register (such as `a7` here[4]) can be loaded with the identity number of the service required; this is used to index the jump table (array) which contains the start addresses of the various service routines. Other registers (typically `a0` ...) can carry arguments for the call.

Add some questions about the various steps?

[3] This is not the only way to handle the situation but it is fairly neat.
[4] `a7/x17` is used for this purpose in Linux etc.

### 14.2.3 ECALL processing

"Thou shalt not corrupt the user's state."

Like any procedure a `ECALL` service routine will have a function, some input parameters and some output results. These should be clearly defined (this is what the comment field is for, okay?). You might decide that your `ECALL`s follow the RISC-V ABI – but bear in mind that other traps (later) will need to preserve *all* the registers. This is what the stack is for!

Note in particular that an **`ECALL` which calls a procedure** (`CALL`, `JAL` ...) must first preserve the pre-existing return address (`ra`) and an **`ECALL` which calls another `ECALL`** must first preserve other information {`MSTATUS`, `MEPC`} as well.

This is probably inviting extra work. Ask, if you are uncertain about this (but really want to try); it's not necessarily ... 'recommended'!

**Nesting ECALLs**

Firstly this is probably not *necessary*. Code can be structured so that the same routines can be called 'internally' if desired. If you do do this it's important that all the return information is preserved so that the original context can, eventually, be restored. Usually this is done by pushing it onto a stack.

On RISC-V a user call (using `JAL` or `CALL`) may need to preserve the Return Address (usually stored in `RA`/`x1`) value before a nested `CALL` overwrites it. An `ECALL` will also need to do something similar. In addition, if `ECALL`s are nested the both `MEPC` and `MSTATUS` must be preserved before the second `ECALL` occurs – the latter to preserve the saved privilege. They can also be pushed onto the stack.

Note that an `ECALL` from machine mode will indicate a different `MCAUSE` and may need to be handled slightly differently – e.g. *not* changing its stack pointer on entry.

Probably easier *not* to nest `ECALL`s unless you are feeling confident in all this!

### 14.2.4 ECALL exit

The `ECALL` instruction has changed[5] the privilege status, generously saving a copy in the `MPP` field of `MSTATUS`. When exiting from the `ECALL` this must be restored. This can be done with the instruction:

```
MRET
```

Other operations will also be needed, undoing any register pushes and recovering the application's `SP`. See also the code on page 63 (and, later, you may want to refer to the ISR exit on page 87 too).

All the different kinds of `ECALL` will have their own service routines. Using the dispatcher above it is possible simply to return at the end of a service routine; however it is worth considering exiting through a short section of return code which is *common to all the service routines*. This is less efficient (i.e. more instructions needed in jumping into this code) but may provide lower maintenance in the future if, for example, the `ECALL` entry routine is changed[6].

[5] Usually: it is possible to '`ECALL`' from any mode though.
[6] E.g. a different set of registers is pushed.

**Bennett tip**

As programmes get longer it gets inconvenient to try to follow their progress manually. Instead of entering the address in a memory sub-window you can use a register name; the window will then update to start at this pointer and move if the register value changes.

An expression can be used instead, thus (say) "`PC - 8`" will show the last few instructions as well and is a convenient way to follow programme execution.

Similarly, `SP` (a.k.a. "`x2`") may be used to follow the stack pointer etc.

### 14.2.5 Terminate programme

An `ECALL` can be used to declare the completion of a user programme. Obviously this should not return. For now it is adequate to have a simple 'loop stop' (i.e. "`J .`"[7]) to 'suspend' execution. In a multi-tasking system, other processes would take control.

Another mechanism (in the lab. system) is described on p. 79.

## 14.3 Practical

Modify your "Hello World" programme to run as an 'application' within a primitive operating system (OS). The 'OS' should initialise itself (including the supervisor stack) and anything the application might need (specifically the user stack) before dispatching control to the user code. User code should (must!) reside in the application memory whilst the handlers should be in the OS 'segment'. The user code should communicate with the peripherals using operating system calls (i.e. `ECALL`s).

Calls which may be useful *now* (more are coming) might include:

- Print character.
- Terminate programme.

Clearly a 'print string' call makes the user programme somewhat trivial (you might like to print more than one message to bulk out the code) however it may be useful in future. If you do provide it note that it is sensible (structured!) for it to call the 'print character' routine to save code duplication.

Later we could, by modifying only the print character routine, redirect the message to a completely different place.

(The programming hints on page 42 may be useful here.)

### 14.3.1 Print string

Well-structured software does not duplicate code: therefore printing a string should invoke the 'print character' routine.

Is printing a *string* worth its own `ECALL`? Opinions vary, although there is an argument that it doesn't implicitly need additional hardware access.[8] Maybe it is more suited to be a user-mode **library** function?

### 14.3.2 Source file organisation

You are going to be using `ECALL`s and similar in future exercises here, so it is worth considering separating your source files to reflect the overall structure to make it easier to reuse (& expand) the code for later exercises.

[7] '.' is interpreted as 'this instruction' by the assembler.

[8] A counter-argument is that efficiency is reduced if every character output requires the rigmarole of OS entry and exit.

# 15 | Exercise 5: Counters and Timers

## Objectives

- Introduce the timer as a peripheral device
  - Some more peripheral programming
- Frequency division by software
- Cursor control on the LCD module

When operating under real-time[1] constraints it is important to be able to measure the passage of time. Time *can* be measured (and delays imposed) by totalling the time it takes a section of code to execute, but this is a very poor way of producing a delay because:

- Execution time may vary, for example due to cache hits or misses
  - this may be very unpredictable
- Interrupts (see later) could insert extra 'invisible' delays into the code
- If in a multi-tasking environment the code could be suspended for an arbitrary time
- The code may be ported to a machine with a different clock frequency

Thus if a real time reference is needed the programme must have access to a separate, fixed-frequency clock which will not be affected by other system activity. This is normally provided by a hardware peripheral known as a 'timer'.

**Definitions**

**Counter:** a hardware circuit which increments (or decrements) according to an external stimulus.

**Timer:** a counter circuit which counts the pulses of a regular, clock input

**Prescaler:** a divider (modulo N counter) used to reduce the number of counts a timer needs to make

A single peripheral often functions as both a counter and a timer (figure 15.1). In practice timers are far more common than simple event counters, so the following description applies primarily to them. A timer can be seen as a subsystem which counts a known (but, likely, programmable) number of clock pulses to indicate a defined interval. Because clocks in computer systems are often significantly faster than that required a prescaler – typically dividing by a power of two[2] – is often an option to slow the counter down to a 'sensible' rate. The result is available to software and sometimes 'pinned out' to provide a reference for other hardware.

[1] i.e. the response of the system must meet some real, worst-case constraints.
[2] A good example is the common '32 kHz' "watch" crystal: true frequency is 32.786 kHz or $2^{15}$ Hz.

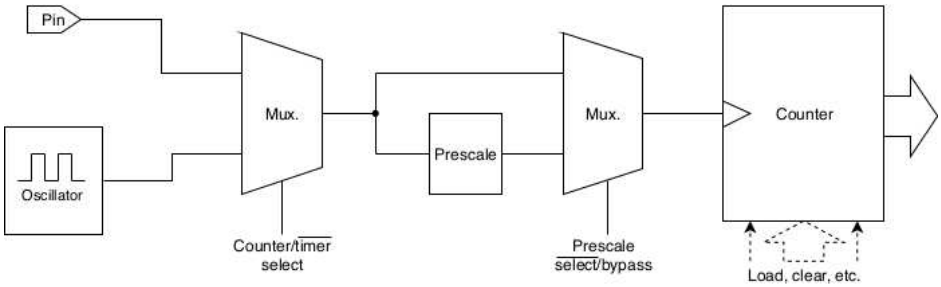



Figure 15.1: Typical counter/timer subsystem

## 15.1 Timer peripheral

**This timer peripheral device is entirely independent from the RISC-V 'built in' timer.**

The timer peripheral (base address `0001_0200`) implemented here is a 32-bit device (only use word operations) with a 32-bit up counter. The clock is prescaled so the counter runs at 1 MHz (i.e. a 1 $\mu$s period) when counting.

The timer can be programmed in different modes of operation by the setting of its mode register: for convenience, mode bits written to together or be set or cleared individually using the related mode set/mode clear addresses. The registers are listed in table 15.1 and the status bits are summarised in table 15.2; there is a more comprehensive description in table 3.11.

**Note that the timer registers are only correctly accessible as words.**

| Offset | Register | Writing ... | Reading ... |
|---|---|---|---|
| 00 | Counter | Counter value (but generally don't!) | Counter value |
| 04 | Limit/modulus | Modulus - 1 | As written |
| 08 | Unused | — | — |
| 0C | Control/status | Status bits | Status bits |
| 10 | Control clear | '1's clear corresponding status bits | — |
| 14 | Control set | '1's set corresponding status bits | — |

Table 15.1: Timer register map

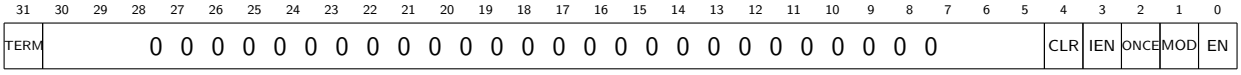



Table 15.2: Timer control/status register bits

Control bit 0 (table 15.2) is the master control and turns the counter on and off. With no other control bits set the counter will run freely through $2^{32}$ states (~71 minutes). This cycle may be curtailed by setting and enabling the limit register which will cycle the counter back to `0000_0000` after that value is reached. Note that this means the limit value is *one less* than the modulus: e.g. **for** a **modulo 4** counter the limit should be **set to 3**: {0, 1, 2, 3, 0 ...}.

A 'sticky' status bit (p. 22) is set when the counter *enters* the `0000_0000` state; the counter can continue or halt at this time, depending on mode. This 'terminal' status bit is exposed in bit 31[3] of the control/status register and can be cleared by writing a `0` there directly or (usually more conveniently) using the 'control clear' register. In the latter case the bit can also be cleared by 'writing' a `1` in bit position 4: this is a more convenient (faster) match to the RISC-V immediate operand coding.

[3]The 'sign bit' when the word is loaded so it can be tested with a single RISC-V branch operation.

Another control bit converts the timer into a 'one shot' mode where it halts (disables) itself when reaching the `0000_0000` state.

The status bit can be made available to the interrupt controller by enabling its output in the control register. When you want to enable timer interrupts (chapter 17) don't forget there is *another* enable bit at the interrupt controller input!

**Types of Timer**

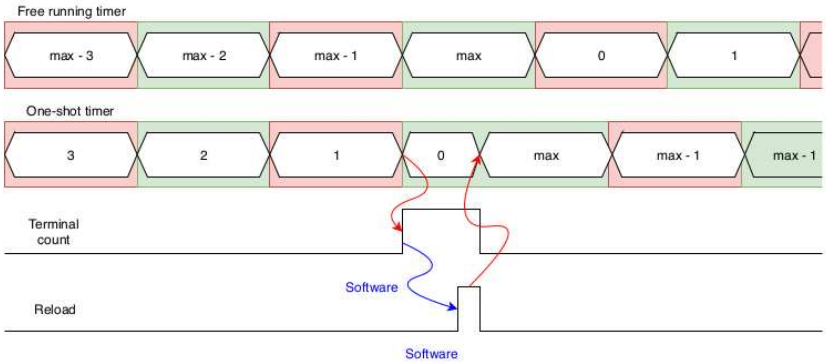
Because the hardware required is relatively simple, timers are often supported in hardware. There are several possible implementations, some examples of which are:

**Free-running:** a value is incremented or decremented on each clock pulse. This integer will have a fixed length (typically 8- or 16-bit) and will cycle modulo its word length. Usually the value will be readable by software and thus an interval can be calculated from the difference between the current and previous readings (beware the wrap-around). Note that there will always be some uncertainty due to the clock resolution, but this is not cumulative because the timer runs freely. Sometimes the counter will be writeable by the processor, although this may introduce cumulative errors.

Often free-running timers have one or more comparison registers; these can be set up to a value which the counter will reach in the future and will produce an output (usually an interrupt) at that time. The lab. board has such a system.

**One-shot:** a slightly more sophisticated timer which counts a preprogrammed number of clock pulses and then signals the fact and stops. Often the counter will count down and stop at zero; usually it is able to provide an interrupt when it has done so. This is useful when some activity is required a known time after an input; for example timing the ignition in a car engine at a known, but probably variable, time in the engine's cycle.

The figure below shows both a free-running and a one-shot timer in 'continuous' use. In each case the timer requests attention and waits for the processor to detect this and service it. Note how the free running timer is able to maintain a regular rhythm whereas the one-shot loses time whilst waiting for the processor.




**Reloadable:** similar in function to a 'one-shot' but they do not stop at zero; instead they load themselves with a value retained in a separate register. This means that they can be programmed as modulo-N counters and then can be left to run freely. (The count can, of course, be changed.) They have an advantage over 'one-shot' timers in applications where time should be measured continuously in that they continue without processor intervention.

As well as providing processor inputs such as regular interrupts reloadable counters are used for programmable clocks within the system. For example one such timer can be used to divide a system clock down to the **baud** (q.v.) clock used on a serial line. This subsequently runs without further (direct) interaction with the processor.

## 15.2 Practical

Write a *stopwatch* programme which increments a value on the display once per second using the hardware timer as a reference. Use buttons on the board to control this these such that one button on starts the count and another[4] causes it to pause. A third button can reset the counter to zero when the count is paused. Hint: a **state diagram** might aid the design here.

Your clock *must* be as accurate as the on-board oscillator which will be within a second or two per day, so take care; we *will* check! Note that writing to the actual counter will introduce inaccuracy.

It is *suggested* that you store the count in BCD (Binary Coded Decimal) to make printing easier — just use a standard hexadecimal print routine (section 15.3). Alternatively a binary-to-decimal conversion routine is provided for download from the module's Blackboard site.

The timer is (of course) in protected I/O space; your application (with all the 'elaborate', parts like checking buttons and printing) should be 'orchestrated' in user mode. To provide 'hardware abstraction' — i.e. to divorce the software from the particular hardware used — the timer should only be programmed and read using system calls (i.e. `ECALL`s). This means that if the programme is ported to a different system only the `ECALL`s need to be changed.

The particular function(s) of your `ECALL`(s) is for you to decide. Try to think of simple, generally applicable functions which many applications might exploit.

In this exercise the counter can be **polled** until a desired state is reached. This is, of course, inefficient; the following exercise (chapter 17) changes the mechanism to use an *interrupt*.

### 15.2.1 Submission

You should submit this exercise for assessment and feedback. The submission should include access to the timer using system calls, plus any appropriate techniques from earlier exercises.

## 15.3 Hexadecimal Print Routine

Probably the easiest way to print a number (in, say, `A0`) in *hexadecimal* is to first assume a routine to print the least significant digit. This is then called the required number of times with the appropriate value shifted/rotated into the correct position. Try it on paper, first.

```
PrintHex8       Make a copy of the input
                Shift right 4 places
                CALL PrintHex4
                Restore value from the copy
                CALL PrintHex4
                RETURN

PrintHex4       Mask off everything except lower 4 bits
                IF A0 > 9 then add 'A' - 10
                ELSE add '0' (ASCII conversion)
                CALL PrintCharacter
                RETURN
```

Note: the ASCII '6' (e.g.) is not the byte 0x06, but the byte 0x36. See p.113 for a complete ASCII table.

This may be a good time to look again at calling conventions (section 3.3, revisited in chapter 16) although here you are free to define your own standard.

[4] Using a single button to toggle start/stop introduces problems which we will cover in a later exercise.

**Binary Coded Decimal (BCD)**

Binary Coded Decimal is a representation where decimal digits are coded into bit fields. Because there are (obviously!) 10 decimal digits each digit uses four bits. Thus, a byte containing binary `0010_0011` would represent $23_{10}$ rather than $23_{16} \equiv 35_{10}$.

BCD was once widely used to avoid division operations for display purposes. Division is quite a complex/slow operation to perform.

## 15.4 Decimal Printing

The difficulty with printing in decimal is that the number to be output must be divided down first. (This is also true in hex, but dividing by 16 is much easier!) The basic RV32I does not have specific division (or multiplication) instructions thus requiring these to be coded in software, which is rather slow. The processor available here has the '`M`' extension (i.e. RV32IM), so does include '`DIV`', '`REM`' and '`MUL`' instructions. (All of these operations – particularly divisions – are still slow to execute, though.)

Assuming division is available, here are two different ways of printing a 16-bit unsigned number in decimal. Step through these (on paper) to see how they work.

### Decimal print — method #1

Start with a table representing the digit positions; for a 16-bit number (0-65535) this would be:

```
DEFW    10000   ; Note these are decimal
DEFW    1000    ; numbers
DEFW    100
DEFW    10
DEFW    1
```

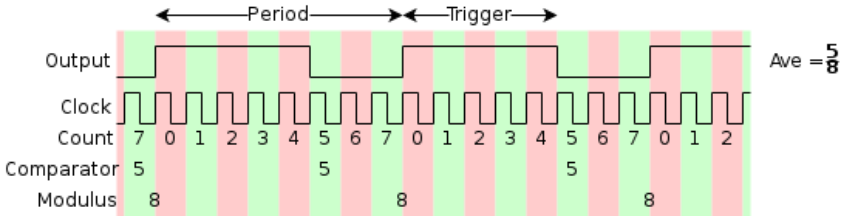
Divide the number to be printed by the first entry in the table, keeping both quotient and remainder. Print the quotient, which must be a single digit. Repeat the process successively using the remainder as input with subsequent lines until the end of the table.

### Decimal print — method #2

Divide the number by 10; stack the *remainder* and repeat with the quotient until the number reaches zero. This pushes the digits to be printed onto the stack in reverse order. Now pop and print the stacked digits so they appear in the correct order. This method has the advantage that it will work on an arbitrary sized number; it is also easier to suppress leading zeros if that is required. If you try this, make sure that the number 0 prints correctly though! On the other hand it is a bit more awkward if you want to print in a fixed-length text field.

**Pulse width modulation: an example of a timer extension**

Pulse Width Modulation (PWM) is a simple means of Digital to Analogue Conversion (DAC) used in a wide variety of applications (q.v.). For example, suppose you want to control the (apparent) brightness of an LED which you can only switch on or off: if you 'flash' the LED sufficiently fast that the eye cannot see the individual flashes – i.e. it *integrates* the output – then the ratio of 'on' and 'off' times affects the apparent brightness. (Note that the eye is not a linear sensor: 'on' for half the time will appear much more than 'half as bright'.) This technique is very widely used in control systems such as motors et al.




Timer(-like) components can be extended with both a programmable cycle length and a comparator which indicates which part of the cycle the counter is in. The comparator output can be wired to an external device and control achieved through varying its value.

**RV32 System timer**

This is completely separate from our timer peripheral.

RISC-V systems should have a *system-wide* timer as part of their architecture. This is a single (64-bit) (even on 32-bit machines) up counter which runs at a constant rate and is shared across all processor cores in a multi-core implementation. As such this timer ("MTIME") is **memory mapped** although it is available for *reading* via CSRs 'TIME' & 'TIMEH'. There is also an accompanying 64-bit 'MTIMECMP' register (*not* reflected by CSRs); when MTIME equals or exceeds (unsigned) MTIMECMP the machine timer interrupt is asserted.

There is a tacit assumption that the 64-bit timer is large enough and slow enough that 'wrap around' will not be a problem: $2^{64}$ $\mu s$ $>$ half a million years.

For reference (only) we've located the memory-mapped timer in our system control peripheral (3.5.2).

This should not be of any concern in these exercises.

**How to stop the processor from a programme**

The processor's behaviour under Bennett is normally controlled by the 'front panel'. However, occasionally, you may want to a programme to stop the processor itself.

To allow for this a port at address `0x0001_0700` is provided; a write to this port will immediately halt execution; the value written is irrelevant.

# 16 | Calling Conventions

To enable routines — possibly built by different tools or even in different languages — to work together a common interface standard must be employed. This is simply a software convention which programmers or compilers are expected to obey.

For example, you could decide that all processor registers will hold the same values on exit from a '`CALL`' which they had on entry; this would usually entail saving (probably stacking) and restoring the values so that the routine could use registers internally. There is a cost in time and energy in each operation.

There is no compulsion to use any particular convention in this lab; this is simply mentioned for future interest.

## 16.1 'Classic' Compiler Calling Convention

Figure 16.1 shows a loose interpretation of the sort of **stack frame** layout a complier might use. (This omits some features such as links to other stack frames which may be included.)

It is assumed here the stack grows 'down'. In this view all the arguments are passed on the stack; see section 16.2 for a slightly different view.

First the arguments are pushed to the stack by the calling routine. The call is then made which might (e.g. x86) automatically stack the return address; alternatively the return address can be pushed as the first act of the called routine (e.g. ARM). The called routine may also push some processor registers to give itself some working space later. Finally the called routine will decrement `SP` further to open up some space for its **local variables**; these will inherit an undefined value of whatever was on the stack previously[1].

When returning, the process is reversed: `SP` is incremented, the working registers are restored to their values for the called routine, the actual return (jump) is made and `SP` is incremented again to get rid of the arguments. That final step can sometimes be done as part of the return – if the language convention and the processor instruction set allow it – or may be done by the caller after the return.

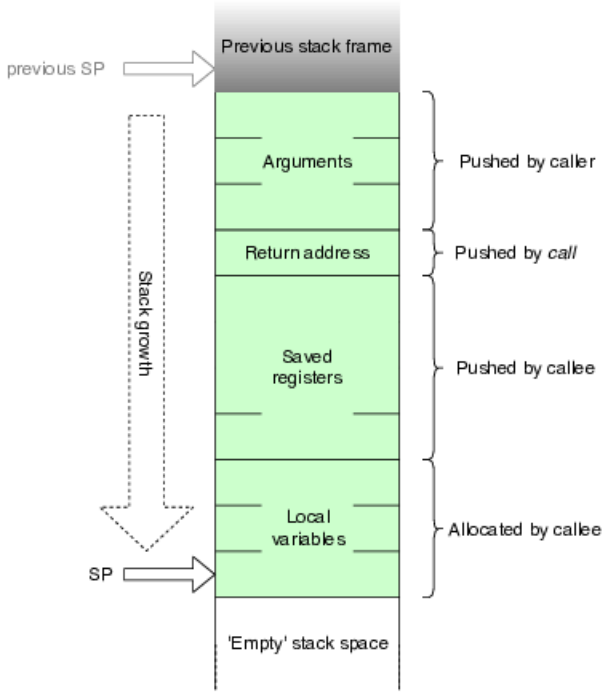



Figure 16.1: Example Stack Layout

[1] Try looking some time

### 16.1.1 Note about C language (etc.)

Some languages insist that procedures are always supplied with the expected number of arguments. C does not do this: think of '`printf(...)`' for example!

This forces a convention that the arguments are pushed to the stack in *reverse* order, so that the first listed argument is the last to be pushed. This means that the first argument is at an offset from the stack pointer determined by the *called* routine and can always be found.

It also means that a calling routine can supply as many arguments as it chooses; if it supplies fewer than expected the called routine can find other (rubbish) values on the stack. Try it sometime.

A further implication is that the calling routine *must* be responsible for *removing* the arguments when the call returns, since the called routine can never be sure how many items were passed. Typically the arguments are not wanted – C does not return modified values – so rather than popping them all (which means expensive memory read operations) a simple add[2] to the `SP` is used.

```
subi   sp, sp, 12       ; Three arguments
sw     s2, 8[sp]        ; Third argument
sw     s1, 4[sp]        ; Second argument
sw     s0, [sp]         ; First argument
call   subroutine       ;
addi   sp, sp, 12       ; Three 4-byte values (in this example)
```

## 16.2 RISC-V Application Binary Interface — Simplified Version

The 'Application Binary Interface' (ABI) is a software specification, used by 'standard' compilers; a few more details are included in section 3.3 or see `riscv.org`. Because languages such as C typically pass (a 'few') arguments into procedures and functions it takes this into account and reserves some '**a**rgument' registers for the job (table 16.1). Because the number of arguments is not bounded it cannot necessarily pass them all in registers although, since there are 'a couple of dozen' uncommitted registers, eight are assigned for this purpose which will satisfy most instances.

The caller can, of course, preserve more registers itself, e.g. by `PUSH`ing them before the `CALL`.

| ABI name | Register(s) | On entry | On exit |
|---|---|---|---|
| zero | x0 | — | Zero |
| ra | x1 | Overwritten by `CALL` | Undefined |
| sp | x2 | Stack pointer | Preserved |
| gp, tp | x3-x4 | Compiler/language management | — |
| s0-11 | x8-x9, x18-x27 | Own variables | Preserved |
| a0-1 | x10-x11 | First & second arguments | Function return value(s) else undefined |
| a2-7 | x12-x17 | Third to eighth arguments | Undefined |
| t0-6 | x5-x7, x28-x31 | Uncomitted | Undefined |

Table 16.1: Simple ABI register model

Thus the example in subsection 16.1.1 becomes the somewhat more efficient:

```
mv     a0, s0           ; First argument
mv     a1, s1           ; Second argument
mv     a2, s2           ; Third argument
call   subroutine       ;
```

If the procedure uses more than eight arguments, the excess are pushed onto the stack before the call ('`CALL`') is made; they must be deallocated after the return.

[2]This is *illustrative*; strictly it breaks the ABI which requires `SP` to always be aligned on a 16-byte address boundary. It should really 'waste' a fourth word.

# 17 | Exercise 6: Interrupts

## Objectives

- Introduce interrupts
- Read buttons to change display
- Invention of user interface

## 17.1 What are Interrupts?

**Interrupts are a mechanism by which hardware can 'call' software.**

The usual model of computer operation is that the user's algorithm is implemented in software. Typically this is broken down into **procedures** which are called to perform ever simpler functions. The lowest level — executing instructions — can be thought of as being implemented in the hardware.

An interrupt is initiated by a dedicated hardware signal to the processor. The processor monitors this signal and, if it is active, is capable of suspending 'normal' execution and running an **Interrupt Service Routine (ISR)**. Effectively it is as if a procedure call has been inserted *between* two instructions in the normal code.

RISC-V systems can have quite complicated hardware interrupt structures. Here they have been (somewhat) simplified (fig 17.1): there are two 'levels' of interrupt controller, one as part of the processor and one as part of the wider system. These are more fully described in section 17.4.

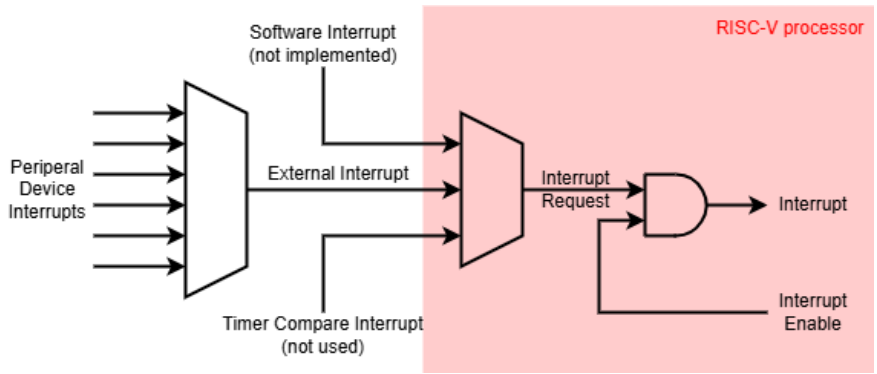



Figure 17.1: Overall interrupt connections

## 17.2 Why use Interrupts?

Most processors will only run a single **thread** (stream of instructions) at a given time. It is often desirable to try to do two or more tasks at the same time (or at least appear to do so). Often this facility can be added with a very small hardware cost.

To illustrate by example think of an office worker writing a letter, a single task. He would also like to answer incoming telephone calls. One way to do this would be to pick up the telephone after typing each line to see if anyone is waiting; he could check to see if anyone is outside the door at the same time ...

This periodic checking is known as **polling** and is clearly inefficient. Firstly it involves a lot of unnecessary effort; most of the time there will be no one there! Secondly a caller could be ignored if our author gets stuck on a particular sentence. Thirdly the 'write letter' process has to 'know' about all the other tasks to monitor.

By adding a bell to the telephone it's possible to allow it to *interrupt* other tasks. Our hero can concentrate on his composition without considering callers, but can still know immediately when one wants attention.

### 17.2.1 Examples of Interrupt Processes

**Waiting.** As has already been seen it is quite common for a processor to have to *wait* for a while. This might be for a fixed time or whilst waiting for external hardware to become ready (e.g. printer off line). In any case polling the hardware is a waste of time the processor could spend doing something else.

**Clocks.** A clock interrupt can provide a regular timing reference whilst other processes are running: more on this below.

**Input.** An interrupt can allow the machine to register an 'unexpected' input, such as the user pressing a key or moving the mouse.

## 17.3 RISC-V Interrupt Behaviour

An **interrupt** may be thought of as a **system call initiated by the hardware**, rather than the application software. As such they can happen at any time and must be 'transparent' as far as the software is concerned.

The RISC-V handles interrupts in the same way as the traps which have already been met — see chapter 12, particularly fig. 12.2 on page 60 — although a few additional details now need to be considered.

Firstly, interrupts have their own set of 'cause' identifiers which are distinguished by having the MSB set in the `MCAUSE` register; this is important because otherwise the 'cause' numbers overlap. These are most easily separated by treating this as the 'sign bit' after loading the value. Thus:

```
csrr    x8, MCAUSE          ;
bgez    x8, exceptions      ; Branch if >= 0  (MSB clear)
andi    x8, x6, 0xF         ; Clear upper bits
...
```

The pre-defined interrupts (after masking has been employed) are given in table 17.1. Further causes may be defined in future and others may be introduced in particular implementations. Here we will concentrate on the Machine External interrupts.

| Number | Reason |
|---|---|
| 0 | User software interrupt |
| 1 | Supervisor software interrupt |
| 2 | *Reserved* |
| 3 | Machine software interrupt |
| 4 | User timer interrupt |
| 5 | Supervisor timer interrupt |
| 6 | *Reserved* |
| 7 | Machine timer interrupt |
| 8 | User external interrupt |
| 9 | Supervisor external interrupt |
| 10 | *Reserved* |
| 11 | Machine external interrupt |

Table 17.1: Interrupt causes

Interrupts are all initially masked so they can be disabled when they might be inconvenient. The RISC-V has two 'levels' of interrupt enable (fig. 17.2): the last level is a bit in the `MSTATUS` register (`MIE`) – bit 3. This bit – initially clear – must be set before any interrupt will be accepted.

When any trap (including `ECALL`) is taken the `MIE` bit is cleared, disabling interrupts, with its previous state copied into the `MPIE` bit (bit 7 in `MSTATUS`). This is restored when the `MRET` is executed.

It is vital that interrupts are disabled when an interrupt is accepted so that the processor only accepts it once: if this did not happen then execution would become stuck, repeatedly trying to enter the service routine. Implicitly the cause of the interrupt must be removed ('acknowledged', in software) before returning from the service routine.

| Register | Abbreviation | Name | Purpose |
|---|---|---|---|
| 300 | MSTATUS | Machine Status | Collected machine status bits. |
| 304 | MIE | Machine Interrupt Enable | Allow individual interrupts. |
| 305 | MTVEC | Machine Trap Vector | Address jumped to on trap entry. |
| 340 | MSCRATCH | Machine trap handler Scratch | Free for software use. |
| 341 | MEPC | Machine Exception PC | Address of instruction which 'trapped'. |
| 342 | MCAUSE | Machine trap Cause | Reason for trap. |
| 344 | MIP | Machine Interrupt Pending | Individual interrupt input states. |

Table 17.2: Relevant CSRs (this augments table 12.1)

Two not-previously-introduced CSRs support the interrupt mechanism: `MIE` allows individual interrupt sources to be enabled/disabled whilst `MIP` represents the states of the different requests (table 17.2). An interrupt will be asserted if the corresponding bits in both registers are set and will cause an interrupt if the `MIE` bit in `MSTATUS` is also set. The bit positions in these registers are mapped to the appropriate causes, so (for example) the machine external interrupt is bit position 11.

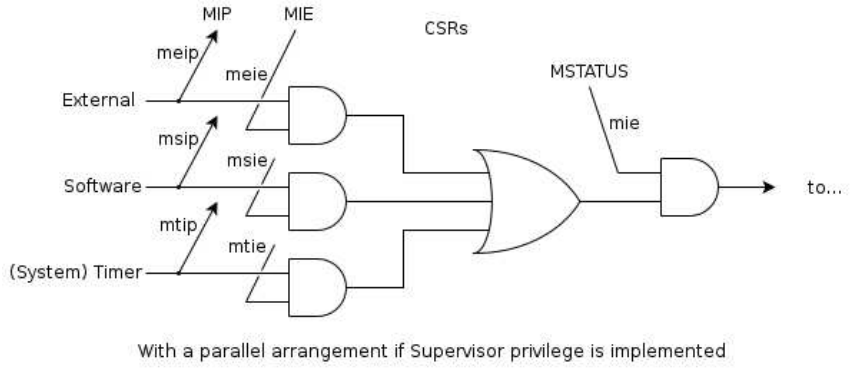



Figure 17.2: Interrupt logic within RISC-V

## 17.4 Interrupt Enable/Disable

It is sometimes desirable to be able to ignore interrupts, both selectively and collectively. An example of the collective case is when the processor is first reset: the states of the various peripherals may be undefined and it is important that the software is not spuriously interrupted before it can initialise them. A second example occurs when an interrupt is recognised since it must only cause the interrupt entry once: interrupts are therefore disabled until the interrupt is properly serviced. In fact **all traps** – including `ECALL`s – disable interrupts on entry since it is likely that an operating system may want to run some 'atomic' (uninterruptable) code. The global interrupt enable bits (RISC-V has more than one) are in the Machine Status CSR (`MSTATUS`) which also keeps their state *before* the trap (see table 17.3 and the box on page 89). (This means the state can be preserved over an `ECALL` etc.) As would be expected, changing the **interrupt enable status is privileged**: user code can't do this.

| 31 | 30-23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 15 | 14 13 | 12 11 | 10 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| SD | — | TSR | TW | TVM | MXR | SUM | MPRV | XS[1:0] | FS[1:0] | MPP[1:0] | VS[1:0] | SPP | MPIE | UBE | SPIE | — | MIE | — | SIE | — |

Table 17.3: Bit map of MSTATUS register (emphasising interrupt control bits)

As the hardware state is undefined at start up these bits are cleared by a processor reset. No interrupts will be serviced until they are cleared in software.

The RISC-V *architecture* has three (or, possibly, more[1]) interrupts defined within the processor. These are labelled: {**external**, **software** and **timer**}; *do not confuse this* system *timer with the timer* peripheral

[1] It is possible to add more, custom interrupts here.

*introduced earlier in this manual!* The logic for collecting these signals resides *inside* the processor and includes selective enables; a sketch of the logic is shown in figure 17.2. In the lab. equipment (as supplied) the 'software' interrupt input – which is for multi-core signalling – is inactive; the (system) timer (p. 79) is present but need not concern us here; all our peripheral inputs are concentrated through the external interrupt signal. The status of these signals is discernible through two CSRs: {`MIP` and `MIE`} (table 17.4).

| | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| MIP | ... | | 0 | 0 | 0 | 0 | MEIP | 0 | SEIP | 0 | MTIP | 0 | STIP | 0 | MSIP | 0 | SSIP | 0 |
| MIE | ... | | 0 | 0 | 0 | 0 | MEIE | 0 | SEIE | 0 | MTIE | 0 | STIE | 0 | MSIE | 0 | SSIE | 0 |

Table 17.4: Bit map of MIP/MIE registers

In addition to these it is usual (in RISC-V systems) to have a separate ***external* interrupt controller**. This unit then feeds into the **external interrupt** input of the RISC-V.

| Offset | Register | R/W | Comments |
|---|---|---|---|
| 00 | Interrupt inputs | RO | Raw signal |
| 04 | Interrupt enables | R/W | |
| 08 | Interrupt requests | RO | After enable |
| 0C | Interrupt mode | R/W | Level/edge |
| 10 | Interrupt edge/clear | R/W | See/clear edge |
| 14 | Interrupt edge set | WO | |
| 18 | — | — | |
| 1C | Interrupt output | RO | Bit 0 |

Table 17.5: Interrupt controller register map

On the laboratory board the peripheral interrupt controller (figure 17.3) is located at address `0001_0400`, There is a register summary in table 17.5 (or see p. 20); it includes enable bits to control the individual interrupt sources which prevent an interrupt request reaching the processor at all and status registers to help identify interrupt sources. All registers should be accessed only as 32-bit words. In addition, some peripherals may contain their own, selective interrupt enables. In most cases, once interrupts are configured that configuration won't often change, although there are times when particular sources may be switched on and off. Globally, interrupts should stay enabled most of the time since they are indicating the need for 'urgent' actions.

The **mode** in table 17.5 refers to the use of the edge detector (figure 17.3): with the mode bit = '`1`' an interrupt input *pulse* is made 'sticky' (p. 22) so it is not missed. The timer interrupt is 'sticky' at source so using a simple level (mode '`0`') is adequate but the facility might be useful when it comes to your own peripheral device(s).

Each interrupt input occupies a **bit position** in the controller: these are repeated in table 17.6.

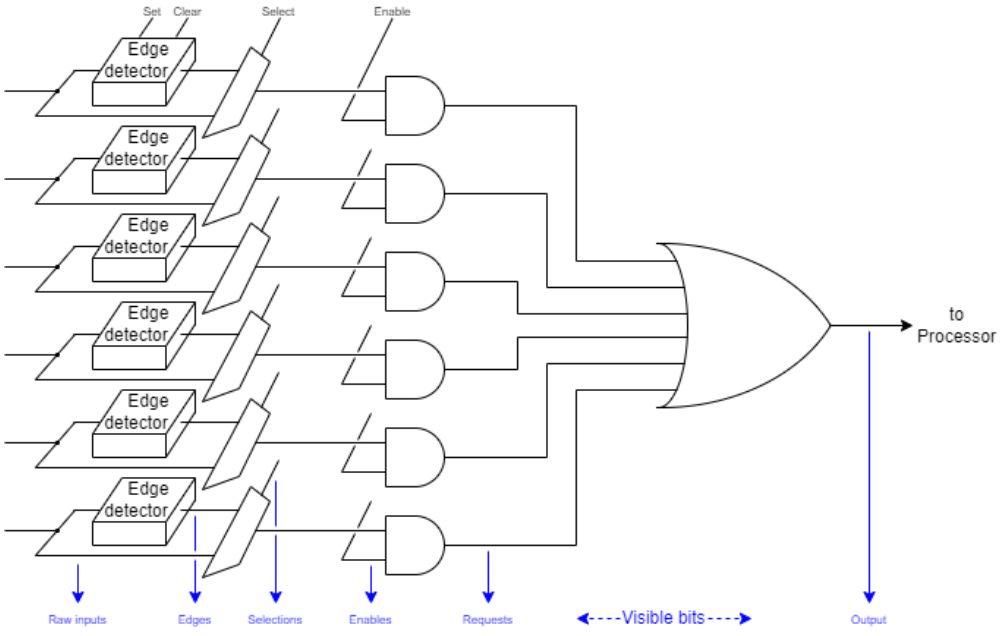



Figure 17.3: Interrupt Controller logic

| Bit | Source | Function | (Recommended) Mode |
|---|---|---|---|
| 0-3 | User #0-3 | Customisable | — |
| 4 | Timer | Timer has reached its terminal count (at least once). | level(/edge) |
| 5 | Button | Top left button has been pressed (at least once). | level/edge |
| 6 | Video | Display vertical sync. | edge |
| 7 | Video | (reserved) | edge |
| 8 | Serial Tx | The serial transmitter is available for writing. | level |
| 9 | Serial Rx | The serial receiver has (at least one) byte waiting. | level |
| 10 | Serial Error | The serial receiver has an overrun error | level/edge |

Table 17.6: Interrupt input bit map (copy)

## 17.5 State Preservation

**"Thou *really* shalt not corrupt the user's state."**

The interrupt entry sequence preserves the minimum of the processor's state necessary; refer to figure 12.2 on page 60 for details. The PC and the mode will have been saved, but that's all. It is essential that the service routine does not change any of the state visible to the user process (for example the contents of t0[2]).

An interrupt service routine will require some 'working' registers and the values in these registers ***must*** first be saved. It is usual to *stack* these values using the stack appropriate for the handler: in this case the machine mode stack. If this is not done the ISR will still work but the user programme will begin to behave erratically as its variables are randomly corrupted.

A fairly easy way to swap stack pointers on a RISC-V is to exploit the (uncommitted) 'scratch' CSR – in this case MSCRATCH. If this is initialised (before interrupts are enabled) with the machine Stack Pointer value it can be swapped in (and out at the end of the ISR) with a "`CSRRW  sp, MSCRATCH, sp`" operation.

This is the same mechanism – the same code – as the ECALL entry section 14.2.1

## 17.6 Trap Disambiguation

Since the RISC-V has only a single trap entry point (to machine mode) all traps arrive at the same address (exception: see box on p. 87). The next job is therefore to separate the different traps.

Interrupts are distinguished from 'synchronous' ('exact') exceptions by the MSB of the MCAUSE CSR. Thus, with MCAUSE in (e.g.) t0 a single instruction:

```
bltz   t0, interrupts     ; Branch if < 0  (MSB set)
```

serves to separate the two. (In practice it may be better to branch to the *other* exceptions since not branching will be a faster path and interrupt latency is probably more time-critical.)

Identifying the particular interrupt is a two-stage process on a RISC-V: first there is the value in the *remainder* of MCAUSE which will indicate the cause at the processor input. Note that, if the 'cause' is left shifted (for example to index into a jump table) the MSB will consequently be lost.

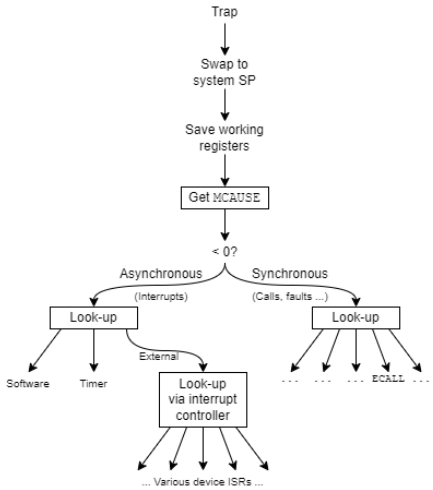



Figure 17.4: Trap entry sequence

There are (currently) three classes of interrupts defined at this level:

**Software** used for inter-multiprocessor signalling.
**Timer** from the system-level external timer, as *visible* in the CSRs.
**External** which is the collection of peripherals in the local processor subsystem.

In our case we will be dealing with **machine** mode **external** interrupts. Having established this, and branched to the next level of handler, it's time to establish which interrupts (remembering that there could be more than one, simultaneously) are active. This involves referring to our own interrupt controller peripheral.

[2]Note that, in this case, the ABI is irrelevant and there is no such thing as a 'temporary' register; *all* state must be preserved!

**Vectored interrupts**

In some RV32 implementations it is possible to *vector* the interrupts (although not the synchronous traps). When this is possible it is done by setting bit 0 of `MTVEC` to '1'. This causes the `MCAUSE` value (stripped of its MSB) to be used as an *index* into a jump table, origined at `MTVEC` (stripped of its LSB).

The upshot is that one of the look-up tables (fig. 17.4) is moved into hardware, reducing the latency of the ISR. The memory *at* `MTVEC` needs to be expanded into a table of jump instructions to forward the PC to the appropriate routine.

The mapping of peripheral interrupts to bits in the controller (`0001_04xx`) is given in table 17.6. The easiest starting point is the requests register (table 17.5) since this filters out disabled signals. After that it is a software job to identify and service an active request.

"If there is more than one concurrent request, is it best to service one at a time or to keep servicing requests until they are all acknowledged?"

Good question! If you expect very frequent interrupts then checking before exiting makes sense since there is a cost entering and leaving the interrupt service routine. On the other hand, if interrupts are expected to be far apart then it may be that simply returning makes sense and, if there is some other active request, starting afresh.

There's no definitive answer. Here, do whatever seems simplest.

## 17.7 Interrupt Service Routine

During the interrupt service routine it is necessary to **acknowledge** the interrupt source. If this is not done then the interrupting device will continue to clamour for attention and will cause another interrupt as soon as interrupts are re-enabled — typically immediately on return from the ISR.

RISC-V systems do not have a *hardware* interrupt acknowledge mechanism so the interrupt requests must be cleared by explicit software operations; this typically involves clearing some 'sticky' status bit. In this system/exercise this is the timer peripheral's terminal count status bit (but beware! there will be a *second* 'sticky' bit in the interrupt controller if you set that input to be 'edge-triggered'; not necessary and not recommended here).

Sometimes peripherals are designed to clear interrupt requests semi-automatically: for example as a *side effect* of *reading* a status register. This can save some code during a (time-critical) interrupt service routine. After some consideration this has *not* been done here since the debugger displaying the register can interfere, rather confusingly. However the serial interface (not explicitly used in any of the practical exercises) does follow the (common) convention that reading the received data register 'removes' the received byte which will reset the receiver status and clear the corresponding interrupt request.

## 17.8 Interrupt Service Exit

When the interrupt service is complete the machine state must be restored. Any registers which have been corrupted must be reloaded by the code itself; finally the interrupt entry sequence must be reversed.

Note that the address saved in `MEPC` is the address of the interrupted instruction; this instruction has not yet been executed. The return is therefore slightly different from an `ECALL` return where (implicitly!) the operation has been done. **After an interrupt return directly to `MEPC`.**

In addition the mode must be restored, interrupts re-enabled etc. This is all achieved with the instruction:

```
MRET
```

## 17.9 Practical

**Part 1:** Experiment with interrupts before running them 'real time' by single-stepping (or 'walking') through an Interrupt Service Routine (ISR). A user button can be used to assert the interrupt signal if it has been enabled. This can then be used to illustrate how an external stimulus can cause a change in a programme's behaviour and you can use this to debug your understanding. There's a checklist of things to be done in table 17.7 (substitute the button for the timer if single stepping).

Note that, if running at full speed, a button press may cause several interrupts due to key bounce. (Don't worry if you don't know about 'key bounce' yet: we'll be covering that in chapter 19.)

**Part 2:** Convert your counter (displayed on the LCD) to an interrupt-driven system. When running this should be independent of any user programme which can be run separately. At the start you may not have a specific 'foreground' application, but an empty loop will do to represent this.

## 17.10 Timer modulus mode

If you haven't done so already it is now sensible to experiment with setting the timer to cycle at a frequency of your choice using its 'modulus' register (section 15.1). Enabling interrupts from the timer will the assert am interrupt at your chosen frequency. The 'sticky' nature of the output bit will keep asserting the interrupt whilst the counter keeps running: it is therefore important to remember to clear the counter when the interrupt is serviced. The exact interrupt latency – the time from interrupt assertion to software response – will probably vary a little but the assertions should occur regularly and therefore overall keep good time, assuming you don't miss any, e.g. by making them *too* frequent for the software.

Since the timer status/interrupt output is 'sticky' the interrupt controller can treat the signal as **level-sensitive**; there is then no need to worry about edge detection in the interrupt controller.

## 17.11 Frequency Division

In principle the timer interrupts arrive at a maximum frequency of 1 MHz (i.e. 1 $\mu$s intervals) – which is really too fast to service in software – and a minimum frequency of about every 70 mins. The counter or clock should only *visibly* increment at 1 s intervals. However there is nothing to say that you can't keep a more accurate clock without displaying the less significant digits!

## 17.12 Advanced

Reformat the output to make a digital clock. Allow the user to set the time using some input buttons (devise your own way to do this).

Don't forget the buttons may 'bounce', so read through the next exercise first.

**RV32 programming puzzle**

Test if the value in s0 is an exact integer power of 2. Other registers may be used.

Target: 2 instructions (quite difficult!)

## 17.13 Debugging Checklist

Did you remember to:

| — Hardware — | | |
|---|---|---|
| Timer | Interrupt controller | Processor CSRs |
| • set the to repeat, modulo limit?<br>• set the correct modulo limit?<br>• enable the interrupt output? | • enable the timer interrupt input?<br>• set to 'level' sensitive? | • enable the **external** interrupt input?<br>• set the appropriate vector (`MTVEC`)?<br>• enable interrupts on the processor (`MSTATUS`)? |
| — Software — | | |
| Before | During | After |
| • add code to the exception traps to: a) separate interrupt from other traps? b) identify 'external' interrupts (`MCAUSE`)? c) identify a particular peripheral interrupt?<br>• (maybe) set up a SP for exception service (`MSCRATCH`)? | • (probably) swap SP to system stack space?<br>• preserve **all** the processor state during the ISR?<br>• clear the interrupt source so it is serviced only once? | • (probably) swap SP back to application stack space?<br>• restore the PC (`MEPC`) to the interrupt position – and *atomically* . . .<br>• . . . restore the mode (and hence re-enable interrupts) when returning from the ISR? |

Table 17.7: Interrupt debugging checklist

The items in grey are advisory, not *strictly* necessary. **Note that there is no ABI rule to allow you to use (e.g.) "temporary" registers without saving: everything user-visible must be preserved.**

**RV32 MSTATUS interrupt bits**

Only one of these bits need concern us here: `MIE` is the Machine Interrupt Enable ('`1`' = enabled).

The other 'M' bits are:

`MPIE` Machine Previous Interrupt Enable (bit 7)
`MPP` Machine Previous Privilege (bits 12:11 – three possible privilege levels {U, S, M}).

which are the the top level of the stack on trap entry and should not be changed unless exception handlers are to be *nested*. (This not advised!) These fields are written to automatically on a trap entry and trap return (`MRET` instruction).

There are equivalent fields for **S**upervisor interrupts – also of no concern here.

# 18 | Shifts and Rotates

These are operations which move the *position* of bits whilst (effectively) retaining their order. This chapter covers the various typical cases, not all of which are available in the (basic) RISC-V instruction set: notably RV32I does not support rotation. You should know they exist, though!

| Name | Typical Abbr. | RISC-V mnem. | Effect |
|---|---|---|---|
| Logical Shift Left | LSL | `SLL(I)` | Move the bit pattern towards the more significant end of the register. Fill at the right with zeros. Lose some bits at the left. (Sometimes also '`ASL`)'. |
| Logical Shift Right | LSR | `SRL(I)` | Move the bit pattern towards the less significant end of the register. Fill at the left with zeros – i.e. an *unsigned* operation. Lose some bits at the right. |
| Arithmetic Shift Right | ASR | `SRA(I)` | Move the bit pattern towards the less significant end of the register. Fill at the left with copies of the original sign bit – i.e. a *signed* operation. Lose some bits at the right. |
| ROtate Left | ROL | — | Move the bit pattern towards the more significant end of the register. Fill at the right with any/all bits 'lost' from the left. |
| ROtate Right | ROR | — | Move the bit pattern towards the less significant end of the register. Fill at the left with any/all bits 'lost' from the right. |

Figure 18.1 illustrates the operation of shift and rotate operations.

Historically, many microprocessors were confined to one-place shift instructions; later implementations might perform multi-place shifts using a succession of single shifts. Typical, modern processors have a hardware unit called a '**barrel shifter**' which will shift by an arbitrary number of places in a single cycle.

Note that, with a barrel shifter, the two rotation operations are complementary so only one is really required.

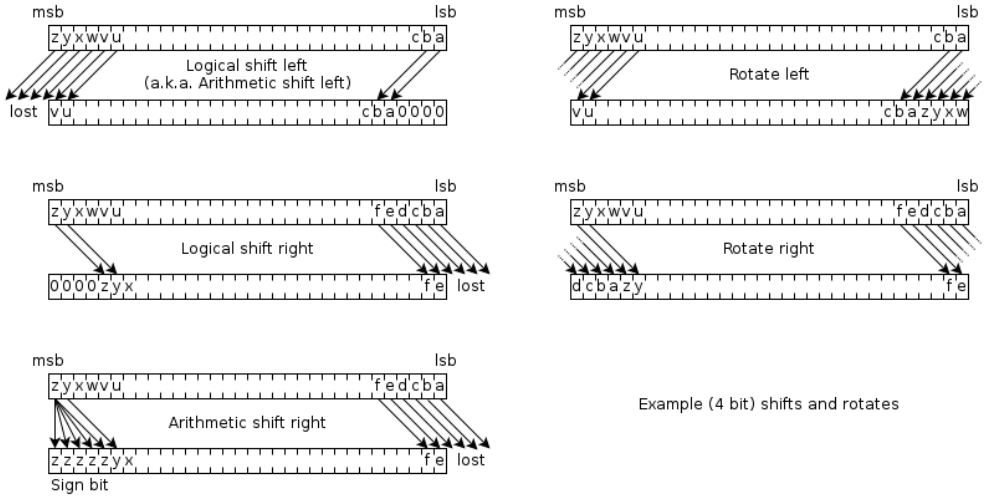



Figure 18.1: Example shifts and rotates

**RISC-V extended arithmetic**

The RV32 has 32-bit registers – but what if you need to calculate with larger numbers?

Probably most processors (including x86, Arm . . . ) use a 'carry flag': an extra bit of storage which can be used to propagate a carry during longer additions and subtractions. This is convenient for the programmer although rarely used for this purpose in practice (it was *much* more important when registers were only 8- or 16 bits wide!) and it is extra state for the hardware engineer to keep track of. RISC-V does not have any such (integer) status 'flags'.

If extended arithmetic is needed a full-size register can be used to represent the carry. Consider the following 64-bit addition of register pairs `S1:S0` to `S3:S2`.

```
add     s4, s0, s2          ; Add lower halves
sltu    t0, s4, s0          ; Synthesize carry state
add     s5, s1, s3          ; Add higher halves
add     s5, s5, t0          ; Add in any carry
```

Here the operation **`SLTU`** (**S**et if **L**ess **T**han, **U**nsigned) is used to set (to 'true' i.e. '`1`') register `t0` if to lower sum apparently yields a smaller value that one of its operands – meaning it must have generated a carry. This is added at the end; if no carry was generated then a 'false' ('`0`') is added instead. The principle can be extended to arbitrarily long operands.

Recoding the example above to preserve `t0` and the equivalent code for subtraction (etc.) are left as exercises.

## 18.1 In other processors

In processors with **status flags** (e.g. ARM. x86) it is common to treat the carry flag as an 'extension' flag in shifts, where it is set to the last-bit-to-fall-off. This is most useful for single place shifts, particularly for extending the number range (e.g. shifting a 64-bit value in a 32-bit processor).

There may also be an extended rotation where the carry is included in the cycle: i.e. a 33-bit rotation encompassing a 32-bit register and the flag. It has some uses . . .

**Multistep bug?**

When interrupts are used 'multistep' sometimes does not count all the steps asked for. This is because a requested step may become an interrupt entry action which is not a real instruction.

Is this a bug? What do you think?

Note that the difference in the number of steps requested and instructions executed gives the number of interrupts taken during the sequence. This may be useful in itself.

**Bennett tip ?????**

In the same way that a `BL` or `SVC` can be treated as a single step, an interrupt service routine can be made transparent to the user. By enabling the relevant option in Komodo ("Active: IRQ") an interrupt will be serviced invisibly, even if the processor is otherwise halted.

This may be useful in debugging 'foreground' code where interrupts must remain active in real time. For example a clock setting programme may be single-stepped even while the clock continues to be updated.

**Loop Unrolling**

Take a piece of code:

```
                LI      t2, 8           ; Count
                LA      s3, somewhere   ; Address
repeat          SW      zero, [s3]      ; Zero memory
                ADDI    s3, s3, 4       ; Move pointer
                SUBI    t2, t2, 1       ; Several times
                BGTZ    t2. repeat      ;
```

Assuming (falsely) no prefetch this executes in 42 cycles — 8 of which are the actual data store cycles. The 'loop overhead' accounts for 16 cycles.

If a prefetch depth of 2 is assumed (typical for many ARMs) the code's execution time is 56 cycles (only the last iteration doesn't waste cycles refilling the pipeline).

If this is time critical code it can be significantly speeded up by **unrolling** the loop:

```
                LA      s3, somewhere   ; Address
                SW      zero, [s3]      ;
                SW      zero, 4[s3]     ;
                SW      zero, 8[s3]     ;
                SW      zero, 12[s3]    ;
                SW      zero, 16[s3]    ;
                SW      zero, 20[s3]    ;
                SW      zero, 24[s3]    ;
                SW      zero, 28[s3]    ;
```

The cost is now 17 cycles, over a 3× speed-up! The penalty is that the code has grown to about 50% longer and would be much longer for more iterations. The speed-up is also less spectacular if the 'body' of the loop is longer. This can be exploited by partially unrolling the loop; e.g. in the above example looping four times and storing two words each time costs only one more instruction but only takes 36 cycles (on this model) — a 1.5× speed-up for a trivial cost.

Loop unrolling (or "in-lining") is normally undesirable, but occasionally useful for small loops in time-critical code. Compilers will sometimes unroll loops at high 'optimisation' settings.

Note that the number of iterations can still be varied ... by calculating and jumping to the appropriate entry point in the code.

**** BEWARE ****

Counting cycles in this manner is a simplistic approach to system timing. Timing can be heavily influenced by other factors, especially memory speed.

For example, if the code is cached, unrolling a loop will use more cache space which may cause more cache misses. As a cache miss could easily cost (say) 100 instruction times, this 'speed' technique could make the system much slower instead.

Branch prediction can be very effective for loops too, so the pipeline flush can often be alleviated.

# 19 | Exercise 7: Key Debouncing and Keyboard Scanning

## Objectives

- Programming around 'real world' problems
- I/O matrix techniques

## 19.1 Key Bounce

Key bounce is a mechanical effect which causes switch contacts to open and close repeatedly as the switch's state is changed from open to closed, or vice versa. The number of bounces and the time taken for the bounce to die out are not deterministic, but the total time of bouncing is bounded. The maximum bounce period depends on the switch being used and the amount of wear it has had, but a typical period would be about 5 ms. This is a short time in human terms, but a long time for a computer. It is therefore generally not possible to simply sample a switch's state and note every change from open to closed as a button press. The switch must be debounced.

In other modules key debouncing has been performed in hardware (generously for you!). This works, but increases the cost of the interface. Software debouncing is a cheaper solution.

## 19.2 Debouncing

Debouncing typically involves waiting for the relevant time; for example it could be noted that a button which was formerly open is now closed and a delay could be imposed before that button is sampled again. This is a fairly simple mechanism to conceive, but requires a time stamp for each button to say when it can/cannot be examined.

Another method of debouncing in software involves the repeated sampling of the key — at decent intervals — but only considering it to have changed state when it has been in a new state for a certain number of consecutive samples. For example a key state could be read as a bit every 1 ms which is shifted into a byte to keep a rolling sample of the last 8 ms of state. The key is 'pressed' when the byte is FF and released when it is 00. For any other state the key is regarded to not have changed.

Another variant of this method is to keep a counter for each key, incrementing it if the key is pressed and decrementing it if it is not. The counter can 'saturate' at 00 and a maximum pre-chosen to indicate when the bounce period has finished. Other variations are also possible.

With any of these methods it is also necessary to note the state of the key before the operation begins. This value changes only when debounce is complete; in the first debouncing method it would be set when the byte reached FF and would remain set until the byte had returned to 00. This provided the necessary hysteresis. When the key is first pressed an event may be generated to cause some effect (such as printing a character).

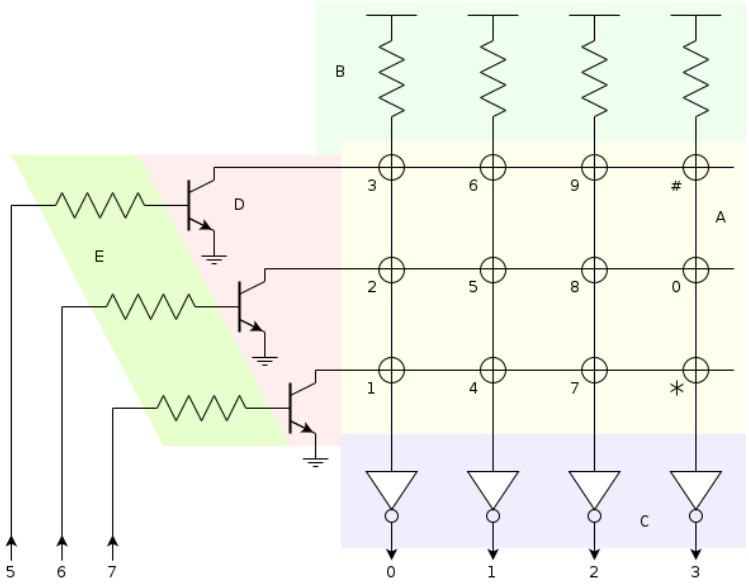



Figure 19.1: Keyboard Matrix

Can you devise a more efficient method to tell if the key has just been pressed? Discuss possibility with the lab. staff (and others) and see if you can find an elegant solution.

The process outlined above needs to be instantiated for every key. The result is a 'map' of the debounced key state which is kept in memory (often sacrificing one byte per key for convenience). This is useful for certain purposes — such as games where key-pressed-or-not is all that is required — but to type into the machine it is also necessary to detect the 'edge' of a key being depressed for most keys[1].

The user interface to the keyboard normally comes down to a "get character" call which waits for the next key press and returns the appropriate code. This is a system call.

In a very simple system (such as here) it may be possible to begin scanning the keyboard on demand. Thus, when input is required, a number of scans (sufficiently separated in time) can be performed and repeated until a key code is returned.

## 19.3 Keyboards

A simple keyboard would have each switch connected to its own input port bit. However for a typical computer keyboard (which has over one hundred keys) this is too expensive in both silicon and wiring.

Instead it is normal to place keys on a two dimensional matrix; 100 keys can be placed on a 10×10 grid and serviced with 20 I/O lines instead of 100, a considerable saving! Figure 19.1 shows the principle of this on a smaller scale. Note: all the inputs and outputs to this circuit are active high. A detailed explanation of this figure is available in an appendix on page 112, if you want/need it.

To scan a row of the keyboard the corresponding input is activated and all others are inactivated. This means that exactly one of the 'horizontal' wires is driven low.

If no keys are pressed then all the 'vertical' wires are pulled high by the resistors. However if one or more keys on the selected row is pressed the corresponding output will be activated via the switch. This provides the state of some of the keys.

A complete keyboard scan is performed by activating each of the inputs in turn (all others are set inactive) and latching the output pattern for each row.

Note that this scan is done electrically and is not affected by the mechanical bouncing of the switch contacts. It is perfectly possible to sample a particular key on successive scans and debounce it appropriately, as described in the preceding section.

[1] Exceptions are keys such as Shift, Ctrl, Alt, ... which act in concert with other keys.

## 19.4 PIO (Parallel Input/Output device)

| Offset | Register | Access | Function |
|---|---|---|---|
| 00 | Data | R/W | Data register |
| 04 | Direction | R/W | Pin direction {'0' = Output, '1' = Input} |
| 08 | Data clear | R/W | Write '1' clears corresponding data bit |
| 0C | Data set | R/W | Write '1' sets corresponding data bit |

Table 19.1: PIO internal registers

The PIO offers simple, bitwise interfacing. Although multiple bits are grouped into a word, each bit can act independently. This 32-bit PIO has four registers (table 19.1).

Reading the data register allows the *pin state* to be determined, whether this is an input or an output (or, *by accident*, there is some form of output 'collision'); writing to the data register will (in the absence of such conflicts) set the *output* bits to the written state. Reading the clear/set locations returns the data register state.

The data direction register controls the pin output enable with a '0' state setting the corresponding pin to drive as an output. This register resets to `FFFF_FFFF` since that is the safest default state.

Writing to the remaining two registers will force a subset – indicated by a '1' in the corresponding bit position(s) – of the data register bits to become '0' or '1' (respectively) whilst leaving the unselected bits alone. This allows bit *changes* to be done *atomically* (in a single operation), avoiding some potential hazards which may occur if a read-modify-write sequence is used. *Reading* either of these locations will return the data register contents, rather than the pin state.

Figure 19.2) shows the bit logic for the PIO, omitting the clear/set logic for clarity. However the direct data register read path is included as well as the bus loading directly from the FPGA pins.

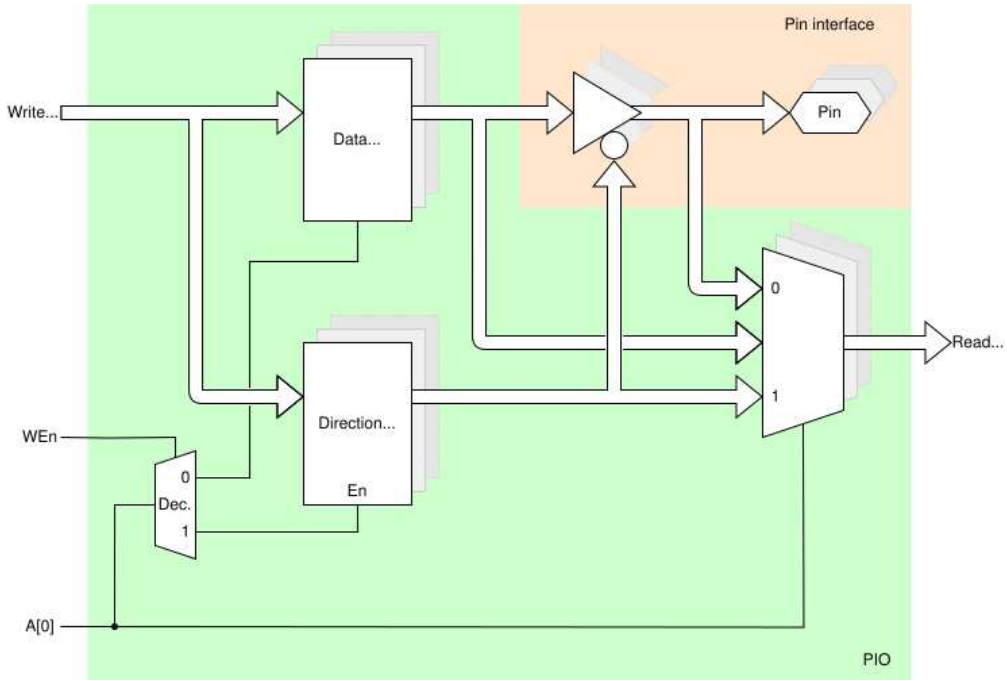



Figure 19.2: PIO Bit Logic

## 19.5 Practical

Attach one of the small, matrix keyboards to the lab. board and use it to input digits which appear on the LCD display.

### 19.5.1 Advanced

Modify your system to make a simple adding machine, totalling and displaying the entered numbers.

### 19.5.2 Submission

You should submit this exercise for assessment and feedback. The submission should use interrupts to scan the keyboard on a regular basis and not rely on software delays.

## 19.6 Key Translation

Note that the 'code' you read by scanning the keyboard is related to the physical position of the key and is not the number written on it. This code needs to be translated to a corresponding ASCII code for printing. For a hint on an efficient way to do this, re-read "Look-up tables" on page 36.

## 19.7 'Real' Keyboard Scanning

In a 'real' system it is normal to scan the keyboard using regular (timer) interrupts (q.v.) to decouple the keyboard from the application programme. During each scan the key states would be read and fed to the debounce software. This would then update the keyboard map as required. This allows characters to be typed even if an application is not 'paying attention', a function known as '**type ahead**'.

If a key has just been pressed (and debounced) the corresponding code needs to be passed to the relevant output stream. This is done by inserting the character into a software First-In, First-Out (FIFO) buffer. In a typical keyboard driver the last key pressed and the time it was observed would also be recorded to allow **auto-repeat**, i.e. the last key sent can be sampled again and inserted again if it is still pressed (i.e. held down) at a certain future time.

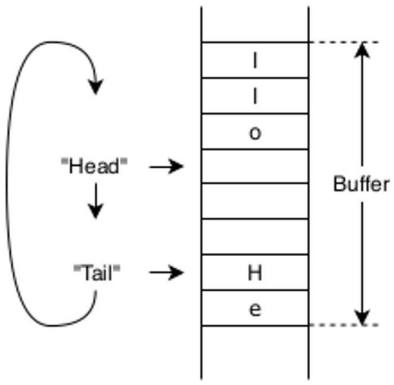
Another function usually provided by the keyboard driver is **rollover**. This feature updates the 'last key pressed' when a new key is depressed, even if the first key has not yet been released. This feature allows faster typing for those sufficiently proficient.

> Try it on your workstation! Press and hold a key and then press another, the output character should change. If the second key is released first the first key will not be seen until it too has been released and pressed again.

With keyboards scanned under interrupts the system call to read a key reduces to getting the next character from the FIFO (or waiting if the FIFO is empty) and returning to the application.

**Software FIFOs**

A FIFO (First-In, First-Out) buffer, or "queue" is an 'elastic' pipe, capable of storing items and releasing them in the same order as they were entered.




In software FIFOs are normally implemented as circular (or 'cyclic') buffers, where the data is kept in an area of memory which is logically wrapped around so that walking off one end brings you back in at the other. The head and tail of the queue is indicated by two pointers which obey these rules. This form of buffering is efficient in that the data never have to be moved in memory; if the head and tail were fixed the data would constantly be shuffling down the memory — a very time consuming process.

In operation items (such as key codes) are inserted at the head and the head pointer is advanced (wrapping if necessary). Items (if any is available) are removed from the tail which advances in a similar fashion in the same direction. The buffer management software must keep track of pointers so that the tail cannot overtake the head if the buffer is empty. Furthermore the head must not 'lap' the tail because then items would be overwritten before they have been read.

**RV32 programming puzzle**

Given a data value in `a0` and a number of bits in `a1`, *mask* `a0` so that only the specified number of bits (starting from bit 0) can be significant.

E.g. if `a0` = `0x1234_5678` and `a1` = `10` this should return `a0` = `0x0000_0278`.

(This sort of operation is quite common in bit packing for (e.g.) communications.)

Target: 4 instructions

(If the operation is repeated – e.g. inside a loop – the 'target' *per additional* iteration can be reduced significantly.)

# 20 | Relocatability

It is usually an advantage if code can be executed wherever it is located in memory, i.e. it is **relocatable** or **position independent**; This is sometimes referred to as 'Position Independent Code' ('**PIC**'). Depending on the processor used this is not always feasible; for example a processor may only have jump instructions which transfer control to a fixed address. However most modern processors are fairly unrestricted and, with care, code can be fully relocatable. (Note that this does not imply that code can be moved during execution.)

For example the RISC-V branch and jump instructions are all **relative branches** i.e. the target address is calculated as an **offset** from the current position. This means that code assembled for a particular address can be moved en masse and still run correctly.

The offset has to fit within the instruction so there is some limitation to the branch distance but that should not be a problem here. (The `CALL` pseudoinstruction can expand to two instructions to give it a larger range.)

Other forms of branches are usually position independent too; for example a subroutine return will go back to the calling routine — wherever that was. The exceptions are when an address is calculated or loaded from memory where extra care is needed if position independence is to be maintained.

A potential exception is when an **address** must be inserted explicitly in software. To get a full address into a register there are two 'obvious' alternatives:

```
                LI      t0, label               ; Load Immediate
                LA      t1, label               ; Load Address
                ...
label           ...
```

If the code is built to run at a known address these will load the two registers with the same value. If the code is loaded at a *different* address, `t0` will still be the same value whereas `t1` will have moved with the code (i.e. it is 'PC-relative')[1].

Whilst it often makes no functional difference, a good habit to observe is:

- Use `LI` if the value is not an address (just some value).
- Use `LA` if the address is part of the code area (e.g. a data table, jump table or embedded string). This is *often* read-only data.
- Use `LI` if the address is some separate 'segment' (e.g. a pointer to I/O space).

As well as code, data should be position independent. Most data are stored in one of four places:

- On the stack (dynamic)
- On the heap (dynamic)
- Within the code (static, often read only such as constants, strings, data tables, ...)
- Another static space (e.g. global variables)

[1] ... as far as we are concerned here. The RISC-V documentation describes 'PIC' and 'non-PIC' variations of `LA` q.v.

**RV32 programming puzzle**

Sign extend a 10-bit number in `s0`.

Target: 2 instructions

Dynamically allocated variables are not a worry here; by definition their addresses are defined at **run time**, usually in some space allocated by the operating system.

Data may be embedded in the code. These can be accessed via **PC relative** addressing as in the previous example, so they need not present a problem. (Note however that there is a ±2 KiB limit on the offset from the `PC` to the variable of interest which can encourage the slightly dubious practice of interleaving data items between procedures[2].)

It may be feasible to use this technique for variables as well as constants but this practice is deprecated as it leads to problems in placing the programme in ROM.

The assembler allows the position of the current instruction to be determined by the value ".". Thus, for example, a 'loop stop' — an instruction which branches back to itself — can be written as:

```
        B       .                       ; Stop here!
```

The hardest problem is finding a place in RAM for the static variables which can be accessed globally rather than — for example — relative to the stack pointer. There is no easy answer to this. The commonest solution is probably to dedicate a register to point to a fixed area of memory (allocated by the OS when the programme starts) and use fixed offsets from that. Sadly this 'loses' a register.

## 20.1 Optional Exercise

This code sequence for a jump table was given on page 44.

```
                LI      t0, Table_size          ; Range check first
                BGEU    a0, t0, Out_of_range    ;
                LA      t0, Jump_table          ;
                SLLI    a0, a0, 2               ; Multiply to index words
                ADD     t0, t0, a0              ; Calculate table entry
                LW      t0, [t0]                ; Get target address
                JR      t0                      ; Jump

Jump_table      DEFW    Routine_0
                DEFW    Routine_1
                DEFW    Routine_2
                DEFW    Routine_3
                DEFW    ...
```

This code is not relocatable; the addresses in the table are the absolute addresses of the various routines and so the jump targets will not move with the rest of the programme.

Write some RISC-V code which dispatches from a *relocatable* jump table.

[2]For example, finely interleaving code and data can lead to inefficient cache usage if code and data are cached separately, as they are on many high-performance processors.

**More on relocation**

Something about relocation tables and more sophisticated linking ???

# 21 | Exercise 8: System Design and Squeaky Noises

## Objectives

- Configure and download hardware: build a system crossing the hardware/software 'divide'.
- Introduce system-level buses and address decoding
- Introduce the piezo-electric buzzer

All the exercises up to this point have used pre-prepared hardware. This exercise requires additional circuitry to be designed and integrated with an existing system. To begin with figure 21.1 shows an overview of the FPGA configuration here.

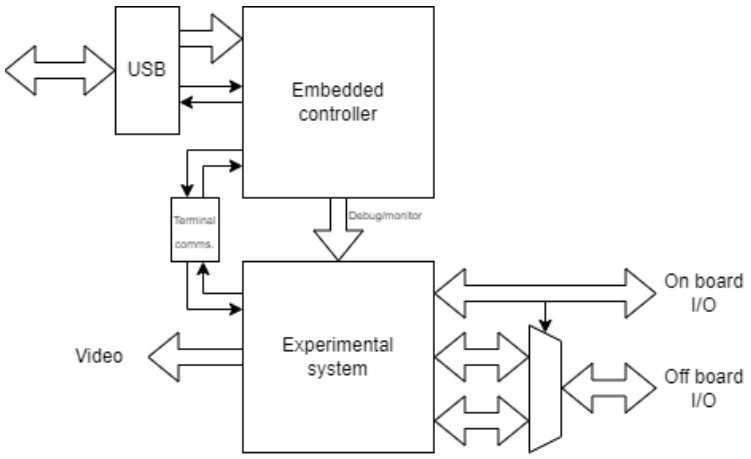


Figure 21.1: System on Chip overview

The embedded controller is another RISC-V system which performs the host communication and controls (starts, stops, steps etc.) the experimental system; it's details are not relevant here. It provides control and visibility of the experimental system, including registers, memory, CSRs and extra features such as breakpoint registers. Additionally it acts as an intermediary for the serial communications so that a terminal can be run across the USB link. All the experimental work is confined to the more elaborate 'Experimental system'.

## 21.1 Experimental RISC-V system

Figure 21.2 shows the architecture of the user RISC-V system. The shaded area encompasses the privileged address space which includes some (system) RAM and the I/O devices. Of the the two user-accessible RAMS, the larger one is an off-chip expansion: all the other components are on the FPGA.

There is an area reserved for user 'I/O Expansion' and this is the space used in this exercise to develop a new peripheral device. A 'placeholder' module exists in the SoC description ready for development here.

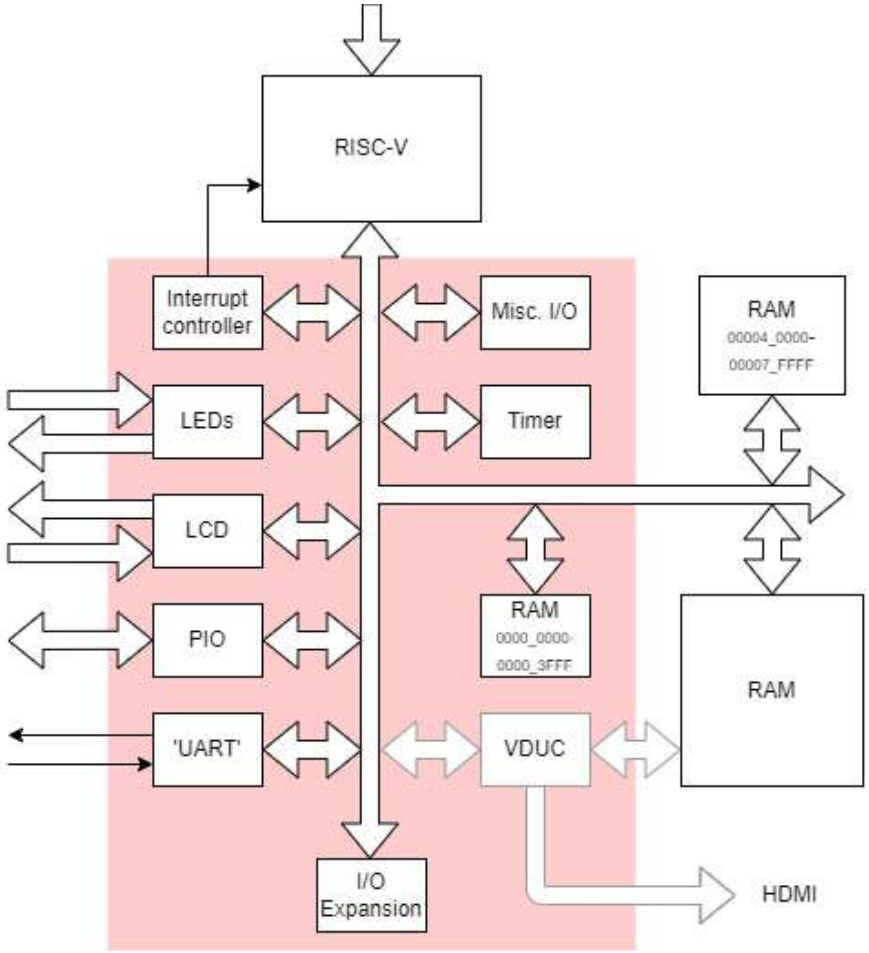



Figure 21.2: User system overview

## 21.2 User peripheral expansion (`user_periph.v`)

The memory-mapped peripheral devices are contained two areas: there is an area (`0001_xxxx`) for those provided with the module and a further area (**`0002_xxxx`**) for your own expansion. This latter area is filled with a module **user_periph.v** which acts as a template for development.

This module implements the simple RISC-V **bus interface** and includes definitions for the **I/O signals**. It is the only module you should need to modify. The bus is already connected to the processor so you don't need to modify anything outside this module.

### 21.2.1 Facilities

The expansion peripheral unit is mapped into 64 KiB of the RISC-V address space (`0002_0000-0002_FFFF`); note that this is a *privilege protected region*. It is unlikely that you will want this many addressable registers – probably no more than a 'handful' – so you can probably ignore some of the (16) input address lines. The region (`0002_xxxx`) is decoded externally and indicated by the `cs_i`[1] input: you should not respond unless this is active ('`1`'), especially to write operations. Note that the addresses are *byte* addresses so the two LSB address bits are not usually very useful.

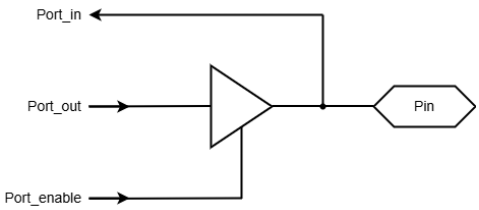
Internally you have access to the system clock input and system reset. There is space to build new bus-based devices within and some possible external connections.

[1] Once upon a time such a 'module' would have been a separate silicon 'chip'; **CS** (**C**hip **S**elect) – sometimes **CE** (**C**hip **E**nable) – are firmly established archaisms.

### 21.2.2 Connection

There are thirty two I/O bits *defined* as a bus, although you probably want to assign (a subset of) them to specific, individual functions. Each signal has *three* wires in its definition:

`port_in` is an input from the relevant connector.
`port_out` is an output *towards* the relevant connector; it must be *enabled* if it is to have a visible effect.
`port_enable` is an output which (if '1') will allow the `port_out` signal to drive the pin.




This arrangement allows bits to be inputs, outputs or bidirectional signals, although it is anticipated that the majority of the signals will be 'hard wired' to a particular direction[2]. No alternate I/O is present in the template supplied, so the outputs have been 'wired off' in a passive state.

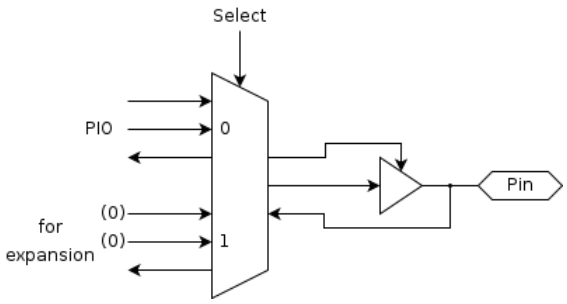



Figure 21.3: I/O pin multiplexing

Although there is usually considerable freedom to create buses 'on chip', getting signals on and off chips is often 'pin limited'; despite advances in packaging technology this is typically a bottleneck. A common solution – and one on offer here – is to *multiplex* pin functions (fig. 21.3) so the user can choose from a selection of different devices. You can find this on many modern microcontrollers including Raspberry Pis.

In this system this is done by software, on a bit-by-bit basis using the 'Pin function' register (`0001_0708`) in the existing system control peripheral (p. 21). In this case there is a choice of the PIO bits or the new expansion.

There are (a rather generous) *four* separate interrupt request lines which are fed to the interrupt controller. You are fairly unlikely to want more than one (if that!) but they're there if needed.

### 21.2.3 Development

It is **strongly recommended** that peripheral development is first simulated and verified: this will save a lot of time and frustration!

A basic simulation test-bench is provided with 'tasks' to simulate processor load and store operations. This can be developed to programme the peripheral(s) in the module, which can be observed and verified from the Questa simulator before integration.

The whole FPGA database is available for rebuilding but you are *strongly* advised to confine your additions to within the 'expansion' module. Also note that rebuilding the FPGA takes a few minutes each time so it's recommended that you **simulate** your device(s) first to save time. The template for a unit test for this unit is also provided.

[2] If an I/O signal is not being used it is usual to make it an *input*; that means accidental external misconnection is less likely to cause electrical damage.

**Bennett tip**

Bennett not only runs your programme but also keeps monitor display updated. This second process will interfere with the smooth running of the 'real' programme. If you try to time your output tones in software you will probably be able to hear this.

Although this 'display update' is enabled/disabled automatically when a programme is run/stopped it is also possible to control this independently using the '**Refresh**' button on the front panel.

## 21.3 Questa Revisited

Create a Questa project for this laboratory: if you've not used Questa here in a previous module such as COMP1211 or COMP2211 (unlikely!) you will need to create the appropriatr space first.

```
mk_dir ~/Questa
```

then copy in the appropriate template material:

```
cd ~/Questa
git clone https://gitlab.cs.man.ac.uk/comp22712_2024/comp22712_<username> COMP22712
```

Hopefully substituting `<username>` with *your* username! You only need to do this once.
The files created will be placed in their own hierarchy: `~/Questa/COMP22712/...`

To start a Questa session use: "`start_questa 227`".

To *reconfigure* the FPGA you will have to quit `Bennett`, synthesise & configure the FPGA from Questa and restart `Bennett`, this time saying '`N`'o to the offer of further configuration.

A correctly synthesized FPGA design will produce a file:

`~/Questa/COMP22712/xilinx_compile/<design_name>.bit`

Remember, any synthesis/error reports may be found in:

`~/Questa/COMP22712/xilinx_compile/<design_name>.srp`

## 21.4 Piezo-electric Buzzers

Piezo-electric crystals are materials which change shape when an electric field is applied[3]. As switching a voltage on an output is a relatively easy thing to do they are widely used by, for example, mobile telephone manufacturers to make irritating squeaky noises. This can be done most simply by grounding one terminal and switching an output bit from 0 to 1 and back to 0 again at the desired frequency. By changing the switching frequency different tones can be played.

A louder noise can be produced by increasing the applied field. With purely digital electronics this can be difficult, but a doubling in amplitude can be achieved by driving both terminals of the device in antiphase[4]. A really sophisticated driver could use voltage steps to output something smoother than a square wave, although this requires considerably more effort.

Note that piezo-electric buzzers are not very responsive at low frequencies so sound 'best' at a few kilohertz. (They also have particular resonant harmonics, which you may hear.)

## 21.5 Practical

Either modify the existing I/O controller (hardware) to include an output to drive the piezo buzzer or make your own driver from scratch (perhaps using `PIO_8` as a template). Make this play a regular tone, or — preferably — a tune. The simplest way to do this is to add an addressable output port (i.e. a latch) and drive the relevant outputs in software, using a system timer. However, at the available frequencies the resolution (1 ms) is poor.

A more capable, if slightly harder (i.e. involving more hardware), method is to use a programmable clock divider which can be set at different frequencies by the CPU and will then generate the tone on its own.

The 40 MHz system clock is available as an input to your unit(s); it's strongly advised that you make everything synchronous with this.

Standard timer interrupts can then be used to control the duration of the note.

[3] They also produce an electric field if they change shape, hence their use in spark igniters, for example.
[4] i.e. at the same time, but in opposite directions.

# 22 | Exercise 9: Project

Choose **one** of the following projects (**or invent** your own) and develop it, as far as you can in the time remaining, using techniques you have learnt in earlier exercises. Concentrate on demonstrating principles gleaned from earlier exercises rather than spending too much effort on elaborate applications code. For example, projects involving user-developed hardware as well as software will be credited more favourably.

Whichever project you choose, use your own imagination to think of ways in which it can (sensibly) develop and try to implement those too. Alternatively, define a project of your own of similar scope. Remember the lab. staff are there to help and advise on project work so whatever you plan to do, talk it over first. Remember that we have cupboards full of various I/O devices available too!

## 22.1 Practical and Submission

You should submit this exercise for assessment and feedback. Submit everything that you have done for this exercise; the more appropriate techniques you can demonstrate, the better. If you have hardware of your own design (including from the previous exercise) you should include this and it will boost your mark.

### 22.1.1 Suggestions

**Calculator** Build a calculator capable of adding, subtracting, multiplying and dividing (and maybe more?) decimal numbers.
If you're short of keys you can add an extra keyboard or use the normal calculator "2nd function" approach.

**Alarm clock** Make a clock capable of displaying (and being set to the correct time). Include an alarm facility which can be set by the user and will 'go off' at the correct time.

**Morse code trainer** Modify your "Hello World" programme to 'print' its output as Morse code.
Extend your keyboard input routines to enable the sending of "text messages".

**Electronic organ** Write a programme which plays notes as specified by buttons on the keypad.

**Dimmer switch** Control the brightness of your LEDs using interrupts and Pulse Width Modulation[1] (PWM).

**Music Player** Write an interpreter ('virtual machine') which reads a list of notes (pitch & duration — possibly volume too) from an input file and plays the tune so described. Remember you will need codes for 'rests' and (hopefully!) 'end of tune'.
For a quick test there are some files available in:
`/opt/info/courses/COMP22712/Code_examples/Tunes/`
Remember that you could add more buzzers to create "harmony".

Hints for writing an tune interpreter:

[1] If you don't know about this, Google will no doubt help.

- Simplify your task by translating notes to frequency/period with a look-up table.
- Note that, as an octave is a factor of two, the frequency can be transposed by an octave with a left or right shift.
- Remember that two notes of the same pitch may need a separator.
- Don't forget a 'silent' setting, for rests and the end of the tune!

**Acoustics for non-musicians**

An "octave" is a factor of 2 in frequency and is divided into eight notes (but is really twelve semitone divisions). In modern tuning — known as "equal temperament" — each of these divisions is the same on a logarithmic scale. This means each semitone has a frequency about 6% different from an adjacent one.

$$factor = 2^{\frac{1}{12}} \approx 1.059$$

Starting a scale at 1 kHz the semitones have the frequencies as shown in the table below. The bold text denotes the notes of the major scale.

The period of the waveform (the time taken to complete a whole cycle) is also given.

| Note | Frequency (Hz) | Period ($\mu$s) |
|---|---|---|
| Do′ | **2000** | **500** |
| Ti | **1888** | **530** |
| | 1782 | 561 |
| La | **1682** | **595** |
| | 1587 | 630 |
| So | **1498** | **667** |
| | 1414 | 707 |
| Fa | **1335** | **749** |
| Mi | **1260** | **794** |
| | 1189 | 841 |
| Re | **1122** | **891** |
| | 1059 | 944 |
| Do | **1000** | **1000** |

Table 22.1: Example of tonic sol-fa

# 23 | RISC-V Assembly Language Mnemonics and Directives

The default assembler is our own. It follows almost all the published RISC-V syntax (as far as we can interpret!) with a few minor exception cases.

- Most notably we use '[' & ']' to specify the base register in loads and stores, rather than '(' & ')'. This is much more usual in assembly languages. The round parentheses are readily confused with arithmetic brackets when evaluating an offset expression and the 'standard' choice seems bizarre!
- `SUBI` is added to be more legible than having to add a negative value.
- Some pseudoinstructions may assemble in different ways for optimisation: for example `LI` will be a *single* instruction if the immediate value allows it and so will `CALL` when the target is in range of a simple `JAL`.
- We've coded the immediate field in the `LUI` instruction differently from (e.g.) `gcc` since this seems easier to work with expressions. To load the constant value `0x1234_0000` we've used: "`LUI S0, 0x1234_0000`" rather than "`LUI S0, 0x12340`" which also seems clearer. In any case you are unlikely to use this: it's easier to stick to the pseudo-instruction "`LI S0, 0x1234_0000`".
- The *directives* (and some expression operators) are our own.

These are example mnemonics as a quick reference guide. This is not a fully comprehensive list see the designers' documentation for a complete guide. Some of these are *pseudoinstructions* – translated into true operations/sequences by the assembler.

**Data Operations**

```
ADD    Rd, Rs1, Rs2
SUB    Rd, Rs1, Rs2
MUL    Rd, Rs1, Rs2
DIV    Rd, Rs1, Rs2
AND    Rd, Rs1, Rs2
OR     Rd, Rs1, Rs2
XOR    Rd, Rs1, Rs2
SLL    Rd, Rs1, Rs2          ; Shift Left Logical
SRL    Rd, Rs1, Rs2          ; Shift Right Logical
SRA    Rd, Rs1, Rs2          ; Shift Right Arithmetic
SLT    Rd, Rs1, Rs2          ; Set Less Than (signed)
SLTU   Rd, Rs1, Rs2          ; Set Less Than (Unsigned)

ADDI   Rd, Rs1, <expression>
SUBI   Rd, Rs1, <expression>
ANDI   Rd, Rs1, <expression>
```

```
ORI     Rd, Rs1, <expression>
XORI    Rd, Rs1, <expression>
SLLI    Rd, Rs1, <expression>
SRLI    Rd, Rs1, <expression>
SRAI    Rd, Rs1, <expression>
SLTI    Rd, Rs1, <expression>
SLTIU   Rd, Rs1, <expression>

MV      Rd, Rs                  ; Move
NOT     Rd, Rs                  ;
NEG     Rd, Rs                  ; Negate

LI      Rd, <expression>        ; 'Load' any value
```

**Memory Transfers**

```
LB(U)   Rd, <expr>[Rs1]         ; Load Byte (Unsigned)
LH(U)   Rd, <expr>[Rs1]         ; Load Halfword (Unsigned)
LW      Rd, <expr>[Rs1]         ; Load Word
SB      Rs2, <expr>[Rs1]        ; Store Byte
SH      Rs2, <expr>[Rs1]        ; Store Halfword
SW      Rs2, <expr>[Rs1]        ; Store Word
LW      Rd, <label>             ; Pseudoinstruction (also LB, LBU, LH, LHU)
SW      Rd, <label>, Rt         ; Pseudoinstruction (also SB, SH) corrupts Rt
```

**Branches**

```
J       <label>                 ; Jump
JAL     Rd, <label>             ; Jump And Link
JR      Rs                      ; Jump Register
JALR    Rd, [Rs]                ; Jump And Link Register
CALL    <label>                 ; Links using x1
RET                             ; Returns to x1

ECALL                           ; System Call
MRET                            ; System Call Return

BEQ     Rs1, Rs2, <label>       ; Branch if Rs1 == Rs2
BNE     Rs1, Rs2, <label>       ; Branch if Rs1 != Rs2
BLT     Rs1, Rs2, <label>       ; Branch if Rs1 < Rs2
BLE     Rs1, Rs2, <label>       ; Branch if Rs1 <= Rs2
BGE     Rs1, Rs2, <label>       ; Branch if Rs1 >= Rs2
BGT     Rs1, Rs2, <label>       ; Branch if Rs1 > Rs2
BLTU    Rs1, Rs2, <label>       ; Branch if Rs1 < Rs2
BLEU    Rs1, Rs2, <label>       ; Branch if Rs1 <= Rs2
BGEU    Rs1, Rs2, <label>       ; Branch if Rs1 >= Rs2
BGTU    Rs1, Rs2, <label>       ; Branch if Rs1 > Rs2

BEQZ    Rs, <label>             ; Branch if Rs == 0
BNEZ    Rs, <label>             ; Branch if Rs != 0
BLTZ    Rs, <label>             ; Branch if Rs < 0
BLEZ    Rs, <label>             ; Branch if Rs <= 0
BGEZ    Rs, <label>             ; Branch if Rs >= 0
BGTZ    Rs, <label>             ; Branch if Rs > 0
```

**CSR transfers**

```
            CSRRW   Rd, <csr>, Rs       ; CSR read/write
            CSRR    Rd, <csr>           ; CSR read
            CSRW    <csr>, Rs           ; CSR write
            CSRS    <csr>, Rs           ; CSR bit(s) set
            CSRC    <csr>, Rs           ; CSR bit(s) clear
            CSRRW   Rd, <csr>, Rs       ; CSR read/write
```

**Miscellaneous**

```
            NOP                         ; No operation
            ILLEGAL                     ; Guaranteed to trap
            LUI     Rd, <expression>    ; Load higher 20 bits
            AUIPC   Rd, <expression>    ; Add 20 higher bits to PC
```

Those last two are clearer when used as part of some more obvious pseudoinstructions such as `LI` & `LA`.

**Directives**

Directive mnemonics are case insensitive.

```
            ORG     $1000               ; Set assembly address
            ALIGN                       ; Next 4-byte boundary
            ALIGN   N                   ; Next N-byte boundary

            DEFB    0, 1, 2, "bytes"    ; Define bytes
            DEFH    $ABCD, 2+2          ; Define halfword(s)
            DEFW    $12345678           ; Define word(s)

label       EQU     value               ; Set value of "Label"
            DEFS    $20                 ; Reserve $20 bytes

            INCLUDE <filename>          ; What it says!
```

**Variable areas**

```
            STRUCT                      ; Begin a data structure
alpha       WORD                        ; offset will be 0
beta        WORD                        ; offset will be 4
gamma       ALIAS                       ; offset will be 8
delta       WORD                        ; offset will also be 8
epsilon     BYTE                        ; offset will be 12
zeta        BYTE                        ; offset will be 13
eta         halfword                    ; offset will be 14
```

**Signed and Unsigned integers**

The RV32 handles 32-bit quantities in its registers. These may be interpreted as signed (two's complement) or unsigned integers (or other things too). Arithmetic operations treat these identically; the processor *doesn't care* what *you* think they are. When it matters then they can be distinguished using different instructions as part of the programme.

It's easy to regard everything as signed integers but this is often poor practice. What about the offset into that table/array? *Can* it be negative? Probably more integer variables are unsigned than signed in a typical programme! Try to use the appropriate 'type'.

Java integers are all signed but some languages (such as C) allow unsigned types.

**Expression Operators**

It is possible, and often valuable, to allow the assembler to work out some values (such as the length of strings and tables). Thus instead of a simple number an integer expression can be used. The following is a summary of the most useful ones.

Unary operators: +, -, ~

Binary operators, highest precedence first:

| | |
|---|---|
| Logical shifts: | {<< LSL SHL} {>> LSR SHR} |
| Logical AND: | {& AND} |
| Logical ORs: | {\| OR} {^ EOR} |
| Multiplication: | * {/ DIV} {\ MOD} |
| Addition: | + - |

**Numbers**

Numeric constants default to decimal (base 10).

Hexadecimal numbers should be prefixed with "$", or "0x".

Binary numbers should be prefixed with ":" or "0b".

Octal numbers — if you really want these — should be prefixed with "@"

## 23.1 Selected CSRs

Table 23.1 lists the RISC-V CSRs which may be relevant to this module. Both the addresses (in the CSR space) and their standard names are included: the assembler will recognise these names.

| Register | Name | Full name | Purpose |
|---|---|---|---|
| 300 | MSTATUS | Machine STATUS | Collected machine status bits. |
| 301 | MISA | (Machine) ISA and extensions | Defines possible/active instruction set. |
| 302 | MEDELEG | Machine Exception DELEGation | |
| 303 | MIDELEG | Machine Interrupt DELEGation | |
| 304 | MIE | Machine Interrupt Enable | Allow individual interrupts. |
| 305 | MTVEC | Machine Trap VECtor | Address jumped to on trap entry. |
| 306 | MCOUNTEREN | Machine COUNTER ENable | |
| 340 | MSCRATCH | Machine trap handler SCRATCH | Free for software use. |
| 341 | MEPC | Machine Exception PC | Address of instruction which 'trapped'. |
| 342 | MCAUSE | Machine trap CAUSE | Reason for trap. |
| 343 | MTVAL | Machine Trap VALue | Trap argument. |
| 344 | MIP | Machine Interrupt Pending | Individual interrupt input states. |
| C00 | CYCLE | Cycle COUNT (low) | Low 32 bits (of 64). |
| C01 | TIME | Real TIME clock (low) | Low 32 bits (of 64). |
| C02 | INSTRET | INSTructions RETired (low) | Low 32 bits (of 64). |
| C80 | CYCLEH | CYCLE count High | High 32 bits (of 64). |
| C81 | TIMEH | Real TIME clock High | High 32 bits (of 64). |
| C82 | INSTRETH | INSTructions RETired High | High 32 bits (of 64). |

Table 23.1: Subset of CSRs of potential interest for COMP22712

# 24 | References

## 24.1 RV32 Reference

Maybe add the local interrupt sources here too?

## 24.2 Abbreviations

**ABI** Application Binary Interface

**CSR** Control and Status Register

**ELF** Executable and Linkable Format

**ESD** Electro-Static Discharge

**FIFO** First-In, First Out (buffer)

**ISA** Instruction Set Architecture

**ISR** Interrupt Service Routine

**LIFO** Last-In, First Out (buffer)

**LSB** Least Significant Bit (occasionally 'Byte')

**MSB** Most Significant Bit (occasionally 'Byte')

**RISC** Reduced Instruction Set Computer

## 24.3 Bibliography

| Number | Reason |
|---|---|
| 0 | Instruction address misaligned |
| 1 | Instruction access fault |
| 2 | Illegal instruction |
| 3 | Breakpoint |
| 4 | Load address misaligned |
| 5 | Load access fault |
| 6 | Store address misaligned |
| 7 | Store access fault |
| 8 | Environment call from U-mode |
| 9 | Environment call from S-mode |
| 10 | *Reserved* |
| 11 | Environment call from M-mode |
| 12 | Instruction page fault |
| 13 | Load page fault |
| 14 | Reserved for future standard use |
| 15 | Store page fault |

Table 24.1: Exception causes

# 25 | Appendix A: Keypad explained

Figure 25.1 repeats the diagram of the expansion keypad. Here is a brief explanation of its parts and operation. The areas of different components are shaded and the letters refer to the key, below.

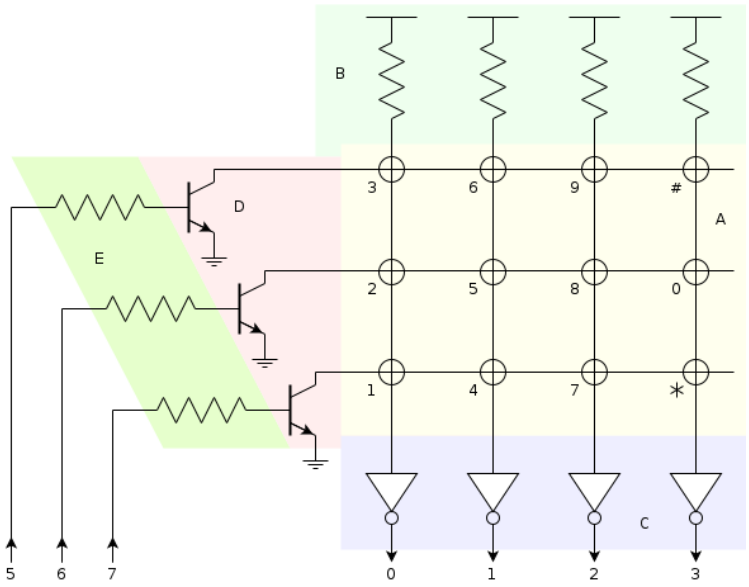



Figure 25.1: Keyboard Matrix

**Area A** is the actual switch matrix. The circles represent the buttons: when pressed the buttons connect the crossing wires together and when released they don't.

**Area B** has *pull-up* resistors: these will usually hold each vertical wire at a logic 'high'. The resistors are quite large so the pull-up is quite 'weak', meaning the value can be overridden fairly easily.

**Area C** you should recognise as digital inverters. These ensure the output is at a good, digital level for input to the FPGA (and provide some protection to that more expensive device). When left alone the input pull-ups (A) will hold the inputs at a logic '1' so the inverters will output logic '0'.

**Area D** has the only *active* components: these are transistors. To be precise these are npn bipolar transistors which operate somewhat differently to the FETs (Field Effect Transistors) you may have met in CMOS circuits. Here, however, they are acting as simple switches.
They are controlled by *current* flowing into the *base* – the terminals on the left. If there is a small current from here to the *emitter* (the terminal with the arrow) then a larger current can flow from the *collector* (identifiable by elimination!) to the emitter and hence, here, to ground. The upshot is a small input current will 'clamp' the output (horizontal wire) to ground. If a switch along this wire is pressed it will also easily overcome the pull-up and thus cause the connected inverter to output a logic '1'.
If no current flows into the base then the transistor is not conducting, the collector 'floats' and pressing a switch will pull up the horizontal wire (which is not visible externally).

**Area E** has resistors to limit the current into the transistors' bases. The inputs are digital *voltages* from the FPGA and there is a limit to the amount of current which should be drawn from there. Current (if any) flows through the transistor(s) to ground: a logic '0' has no driving potential so no current will flow (transistor 'off') whilst a logic '1' will drive a small current set by the resistor value which keeps the current small. The transistor then acts both as an amplifier and a signal inverter.

# 26 | ASCII Character Set

| | | | |
|---|---|---|---|
| 00 | ^@ | NUL | Null |
| 01 | ^A | SOH | Start of header |
| 02 | ^B | STX | Start of text |
| 03 | ^C | ETX | End of text |
| 04 | ^D | EOT | End of transmission |
| 05 | ^E | ENQ | Enquire |
| 06 | ^F | ACK | Acknowledge Idle |
| 07 | ^G | BEL | Bell block |
| 08 | ^H | BS | Back space |
| 09 | ^I | HT | Horizontal Tabulate |
| 0A | ^J | LF | Line feed |
| 0B | ^K | VT | Vertical Tabulate |
| 0C | ^L | FF | Form feed |
| 0D | ^M | CR | Carriage return |
| 0E | ^N | SO | Shift out |
| 0F | ^O | SI | Shift in |
| 10 | ^P | DLE | Data link escape |
| 11 | ^Q | DC1 | Device control 1 |
| 12 | ^R | DC2 | Device control 2 |
| 13 | ^S | DC3 | Device control 3 |
| 14 | ^T | DC4 | Device control 4 |
| 15 | ^U | NAK | Negative Acknowledge |
| 16 | ^V | SYN | Synchronous Idle |
| 17 | ^W | ETB | End of transmitted block |
| 18 | ^X | CAN | Cancel |
| 19 | ^Y | EM | End of medium |
| 1A | ^Z | SUB | Substitute |
| 1B | ^[ | ESC | Escape |
| 1C | ^ | FS | File separator |
| 1D | ^] | GS | Group separator |
| 1E | ^^ | RS | Record separator |
| 1F | ^_ | US | Unit separator |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 20 | | SP | Space | 40 | @ | 60 | ` |
| 21 | ! | | | 41 | A | 61 | a |
| 22 | " | | | 42 | B | 62 | b |
| 23 | # | | | 43 | C | 63 | c |
| 24 | $ | | | 44 | D | 64 | d |
| 25 | % | | | 45 | E | 65 | e |
| 26 | & | | | 46 | F | 66 | f |
| 27 | ´ | | | 47 | G | 67 | g |
| 28 | ( | | | 48 | H | 68 | h |
| 29 | ) | | | 49 | I | 69 | i |
| 2A | * | | | 4A | J | 6A | j |
| 2B | + | | | 4B | K | 6B | k |
| 2C | , | | | 4C | L | 6C | l |
| 2D | - | | | 4D | M | 6D | m |
| 2E | . | | | 4E | N | 6E | n |
| 2F | / | | | 4F | O | 6F | o |
| 30 | 0 | | | 50 | P | 70 | p |
| 31 | 1 | | | 51 | Q | 71 | q |
| 32 | 2 | | | 52 | R | 72 | r |
| 33 | 3 | | | 53 | S | 73 | s |
| 34 | 4 | | | 54 | T | 74 | t |
| 35 | 5 | | | 55 | U | 75 | u |
| 36 | 6 | | | 56 | V | 76 | v |
| 37 | 7 | | | 57 | W | 77 | w |
| 38 | 8 | | | 58 | X | 78 | x |
| 39 | 9 | | | 59 | Y | 79 | y |
| 3A | : | | | 5A | Z | 7A | z |
| 3B | ; | | | 5B | [ | 7B | { |
| 3C | < | | | 5C | \ | ¥ | 7C \| |
| 3D | = | | | 5D | ] | 7D | } |
| 3E | > | | | 5E | ^ | 7E | ~ → |
| 3F | ? | | | 5F | _ | 7F | ← DEL, RUBOUT |

## 26.1 Peripherals — This is a wandering/lost section at present

Embedded controller (fig. 26.1) – omit from published copy.

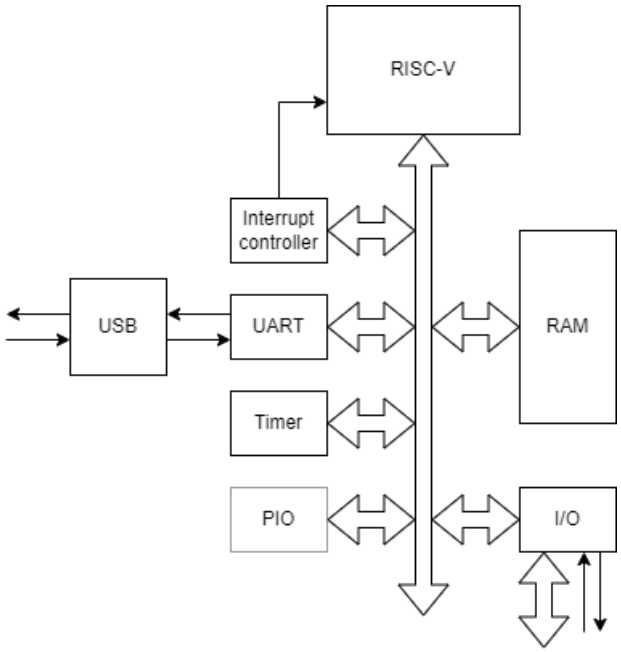



Figure 26.1: Embedded controller overview

The initial area contains a selection of distinct I/O devices as instantiations of smaller modules; it is also possible (and, later, necessary) to alter or augment these. The description of each type of peripheral device is given below.

### 26.1.1 PIO (Parallel Input/Output device)

See earlier?

### 26.1.2 Timer

| Offset | Register | Access | Function |
|---|---|---|---|
| 00 | Timer | R/W | Up counter |
| 04 | Limit | R/W | Modulus |
| 08 | — | — | Unused |
| 0C | Control | R/W | See table 3.11 for details |
| 10 | Control clear | WO | Write '1' clears corresponding mode bit |
| 14 | Control set | WO | Write '1' sets corresponding mode bit |

Table 26.1: Timer internal registers

The 'Control/status' register has only six implemented bits (fig. 3.11). It may be written directly or subsets of the (active high) mode bits can be cleared or set by writing '1's to their bit positions in the appropriate registers. See page 19 for more details.

The counter only counts when enabled. It can be reset *after* reaching the programmed limit: thus to count 100 times the limit should be set to 99 etc. It will then be cleared to `0000_0000` on the next 'tick'. If programmed to 'Repeat' it will continue counting else it will halt. Thus the timer can be used to generate regular interrupts at a programmable rate or produce a single interrupt after a programmed delay.

Bit 31 of status is set when the counter is clocked at the limit (i.e. as it returns to `0000_0000`). This bit is 'sticky' and will remain set until cleared by software. This bit will assert an interrupt if enabled in the control register.

### 26.1.3 UART (Universal Asynchronous Receiver/Transmitter)

This is for the master UART. ** Not for manual! **

| Offset | Register | Access | Function |
|---|---|---|---|
| 00 | Baud | R/W | Up counter |
| 04 | Control | R/W | Modulus |
| 08 | Rx Data | RO | 8-bit data |
| 08 | Tx Data | WO | 8-bit data |
| 0C | Status | RO | <0 >Rx ready<br><1 >Rx framing error<br><2 >Rx overrun error<br><3 > —<br><4 >Tx ready |

Table 26.2: UART internal registers

Fairly standard, if primitive, 8-bit UART. Hard-wired to 8-bits/character, no parity.

The Rx status bits are cleared by an Rx_data read. Rx errors are ORed as a potential interrupt source.

[Probably ought to include reset and Rx/Tx enable bits? – and interrupt enables.]

(fig. 26.2).

### 26.1.4 Interrupt controller

This is for the master interrupt controller.

| Offset | Register | Access | Function |
|---|---|---|---|
| 00 | Inputs | RO | Raw interrupts |
| 04 | Enables | R/W | Enables |
| 08 | Requests | RO | Gated with enable |
| 0C | Mode | R/W | Level or edge |
| 10 | Edge | RO | . . . has occurred |
| 10 | Edge | WO | Clear bit(s) |
| 14 | Requests | WO | Set pending request |
| 18 | — | – | — |
| 1C | Output | RO | In bit 0 |

Table 26.3: Interrupt controller internal registers

(fig. 26.3).

The Spartan-3 FPGA is used to provide the majority of the I/O in this lab. The FPGA can be loaded with a user-selected configuration so that the I/O can be customised for particular applications. However this is not yet necessary as there is a default configuration loaded on start up.

The default configuration has eight, bit programmable 8-bit PIOs which are connected to the central eight connections of each row of the I/O connectors at the front of the board. The keyboard interface should be plugged into the left, front connector which corresponds to PIO #S0.

The FPGA is mapped into an 8-bit wide area of memory occupying the address space 2xxxxxxx. Only byte transfers are sensibly defined. Seven address lines — A[6:0] — are provided and all bytes are significant, so the address space repeats every 128 bytes.

Each PIO is programmed via two registers (fig. 19.2): one holds data which should be output, the other defines the direction of the pin. A bit value of “0” in the direction latch programmes the corresponding bit in that PIO to be an output; a value of “1” programmes the bit in that PIO to be an input. The direction latch resets (at hardware reset) to a value “FF” so that all the pins default to being inputs. This register can be read back.

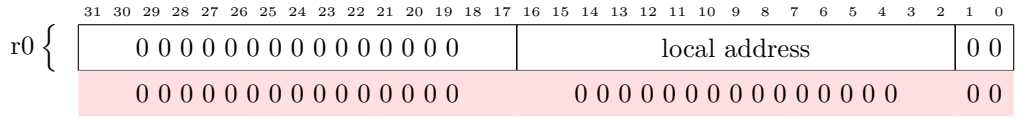
Why is defaulting to inputs (at reset) the sensible behaviour for the PIO bit?

The data register contents is output on any pins defined as outputs. Bits programmed as inputs are ignored. When read this register returns the values at the pins themselves, so input bits should give external input states whilst output bits should be as programmed.

This is just a source for cribbing.

Registers of the RV32I. Based on RISC-V documentation and Patterson and Waterman "The RISC-V Reader" (2017)

https://github.com/riscv-non-isa/riscv-asm-manual/blob/master/riscv-asm.md




Should bits above [1:0] exist?

**Why "Bennett"?**

Bennett's Monitor (Varanus Bennetti) is a **monitor** lizard from Micronesia.

In computing, a "**monitor**" is (amongst other things) software which allows a low-level view inside a machine.

Picture from Wikipedia: Creative Commons.

Credits: Valter Weijola, Varpu Vahtera, André Koch, Andreas Schmitz and Fred Kraus; Thibaud Aronson. Published by the Royal Society.

Bennett is our successor to our previous *monitor* – "Komodo" (named after the famous 'Komodo dragon').