CHERI
Capability Hardware Enhanced RISC Instructions
Platform Reference Manual

Version 1.0

This interim document is not released for public consumption

Robert N. M. Watson, David Chisnall, Brooks Davis,
Wojciech Koszek, Simon W. Moore, Steven J. Murdoch,
Robert Norton, Jonathan Woodruff
SRI International and the University of Cambridge[1]

March 16, 2013

[1]Sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contract FA8750-10-C-0237. The views, opinions, and/or findings contained in this report are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the Department of Defense.

Contents

Abstract

This document is the *Platform Reference Manual* for SRI International and the University of Cambridge's research prototype implementation of the Capability Hardware Enhance RISC Instructions (CHERI) instruction set architecture (ISA). The document is intended to capture our thoughts early in the research and development cycle.

The *Platform Reference Manual* is targeted at hardware and software developers working with the BERI and CHERI prototype CPUs in simulation and synthesised to FPGA targets. The manual includes chapters on the implementation status of the 64-bit MIPS and CHERI ISAs in the prototype, the CHERI Programmable Interrupt Controller (PIC), the CHERI processor implementation, simulating the CHERI processor, the CHERI debug unit, the CHERI test suite, and information on using CHERI with Altera FPGAs and Terasic tPad and DE4 boards.

Acknowledgments

The authors of this report thank other members of the CTSRD team, and our research collaborators at SRI and Cambridge:

Peter G. Neumann	Ross J. Anderson	Jonathan Anderson
Gregory Chadwick	Nirav Dave	Khilan Gudka
Steven Hand	Asif Khan	Myron King
Ben Laurie	Patrick Lincoln	Anil Madhavapeddy
Andrew Moore	Will Morland	Alan Mujumdar
Philip Paeps	Michael Roe	John Rushby
Hassen Saidi	Muhammad Shahbaz	Stacey Son
Richard Uhler		

The CHERI team wishes thank its external oversight group for significant support and contributions:

Lee Badger	Simon Cooper	Rance DeLong
Jeremy Epstein	Virgil Gligor	Li Gong
Mike Gordon	Andrew Herbert	Warren A. Hunt Jr.
Doug Maughan	Greg Morrisett	Brian Randell
Kenneth F. Shotting	Joe Stoy	Tom Van Vleck
Samuel M. Weber		

Finally, we are grateful to Howie Shrobe, DARPA CRASH programme manager, who has offered both technical insight and support throughout this work.

Chapter 1

Introduction

This is the *Platform Reference Manual* for the Capability Hardware Enhanced RISC Instructions (CHERI) prototype. It complements the *CHERI Architecture Document*, which describes the CHERI architecture and ISA, and *CHERI User's Guide*, which provides information on building software for, and using, the CHERI prototype. This document describes the status of the processor ptotype, early information on the implementation itself, and reference information for variouas aspects of the hardware platform, such as the CHERI Programmable Interrupt Controller (PIC) and supported Altera, Terasic, and Cambridge/SRI IP cores.

1.1 Version history

This is the first version of the *CHERI Platform Reference Manual*; some content currently in this document has previously appeared in earlier versions of the *CHERI User's Guide*.

1.0 The first version of the *Platform Reference Manual*, created from both relocated chapters of the existing *User's Guide*, and new content, such as information on the CHERI Programmable Interrupt Controller (PIC), and improvements to the peripheral description (such as addition of a boot loader area to the DE4 Intel StrataFlash layout, and information on the Cambridge HDMI controller).

1.2 Bluespec Extensible RISC Implementation (BERI)

The Bluespec Extensible RISC Implementation (BERI) is a platform for performing research into the hardware-software interface, which has been developed as part of the CTSRD project. It consists of a 64-bit MIPS CPU implemented in Bluespec System Verilog, and a complete software stack based on FreeBSD, Clang/LLVM, and a range of popular open source software products such as Apache and Chromium. Wherever possible, BERI makes use of BSD- and Apache-licensed software to maximise opportunities for technology transition.

1.3 CHERI prototype

Capability Hardware Enhanced RISC Instructions (CHERI) is the first research project built on the BERI platform, and extends BERI to include a capability coprocessor. The first CHERI prototype was developed between 2010 and 2012 by Jonathan Woodruff, based on the 64-bit MIPS implementation originally developed by Gregory Chadwick, and was the foundation for CHERI ISA research; that version is the primary focus of the current *CHERI User's Guide*, which discusses both BERI and CHERI, often using the terms interchangeably. In the future, a stronger differential will be made between the two, and it is likely that the *CHERI User's Guide* will become two different documents, one focused on BERI, and the other on CHERI.

1.4 CHERI2 prototype

CHERI2 was developed between 2011 and 2012 by Nirav Dave and Robert Norton using a highly stylised form of Bluespec in order to better support formal verification through our software that maps Bluespec directly into SRI's PVS. In addition, CHERI2 is designed to support multi-threaded and multi-core operation. As of late 2012, CHERI2 is also able to boot the FreeBSD operating system, but is considered significantly more experimental. Although CHERI and CHERI2 share significant infrastructure – for example, memory subsystems and simulated peripheral busses, we do not envision the two implemetnations converging. Instead, we expect a gradual migration of research to CHERI2 as its more advanced features mature, while keeping "production" development of CHERI features on the first prototype in order to avoid premature imposition of strong stability requirements on a highly experimental implementation. In the longer term, CHERI2's support for formal methods tools should lead to much greater correctness and reliability.

1.5 Getting CHERI

We plan to distribute the CHERI prototype, and reference software stack, as open source. At the time of writing, the prototype is not yet publicly available, but may be requested by e-mailing Dr Peter Neumann[1] at SRI International, or Dr Robert Watson[2] at the University of Cambridge.

1.6 Using CHERI

The CHERI prototype is implemented in the Bluespec hardware description language (HDL), which may be compiled into a C-language simulator, or synthesised for an FPGA target. The former requires access to the commercial Bluespec toolchain; the latter also requires access to the Altera FPGA toolchain.

[1]neumann@csl.sri.com

[2]robert.watson@cl.cam.ac.uk

Currently, there are two supported Altera-based FPGA boards: the Terasic DE4 and the Terasic tPad; the former offers greater FPGA performance and memory capacity; the latter includes a VGA flat panel with touch screen. We are currently in the planning phases for a port of the CHERI prototype to the NetFPGA 10G research platform, which is based on a Xilinx FPGA.

1.7 Licensing

Currently, the CHERI prototype is proprietary to SRI International and the University of Cambridge. In the immediate future, we hope to release an open source version of the prototype under a combination of the Apache open source license (for hardware) and BSD open source license (for software).

1.8 Document Structure

This document is an introduction to and reference manual for the CHERI processor prototype in simulation, and synthesised for Altera FPGAs on Terasic boards:

Chapter 2 enumerates portions of the MIPS and CHERI ISAs implemented by the CHERI prototype; in particular, it documents sections of the MIPS ISA (such as the floating point unit) that have been intentionally omitted. It also documents the implementation status of CHERI-specific ISA features, and aspects of the configuration of reference CHERI systems, such as physical memory maps.

Chapter 3 describes the CHERI Programmable Interrupt Controller, an integrated device supporting interrupts from larger numbers of peripherals than processor interrupt lines, and also used for inter-processor interrupts (IPIs).

Chapter 4 provides more detailed information about the implementation of the CHERI prototype, including a high-level guide to its source code.

Chapter 5 documents how to check out the CHERI source code, build the CHERI simulator, and run the CHERI unit test suite. Various build options, including debug options, are discussed.

Chapter 6 describes the CHERI hardware debug unit, which allows low-level access to processor internals, for the purposes of debugging, via a real or simulated UART.

Chapter 7 documents the CHERI unit test suite, including how to run the suite and add new tests.

Chapter 8 describes how to configure and synthesise CHERI in the Altera development environment.

Chapter 9 describes how to build and synthesise the CHERI prototype for the Terasic tPad and DE4 FPGA teaching boards.

Chapter 2

The CHERI ISA

The CHERI prototype blends concepts described in the *CHERI Architecture Document* with a 64-bit MIPS-ISA CPU implementation. This chapter summarises the 64-bit MIPS and CHERI ISAs implemented in the CHERI prototype.

2.1 The CHERI ISA

CHERI extends the 64-bit MIPS ISA, synchronised loosely with the MIPS 4000 ISA, with a capability coprocessor. The intent of the CHERI prototype is to allow the exploration and validation of the CHERI ISA design. While it would have been possible to implement a new ISA entirely from scratch, we selected the MIPS ISA as a starting point in order to allow us to exploit existing software infrastructure: compilers, toolchain, debuggers, operating systems, and applications wherever possible. This approach allows us to incrementally deploy new ISA features against a known baseline, and demonstrate the realism of the approach.

To this end, we have implemented a subset of the 64-bit MIPS ISA sufficient to execute much existing software, including the open source FreeBSD operating system, X.org window server, and other higher-level application components. Our intent is not to create a complete MIPS implementation: many existing commercial products exist in this space; rather, our goal is to produce a starting point for research into the hardware-software interface. A reasonable comparison might be made between the CHERI prototype and the MIPS 4000 ISA – the first 64-bit version of MIPS. The CHERI prototype implements the following high-level features roughly in accordance with MIPS 4000:

- A pipelined processor design
- 32 64-bit general-purpose registers usable with the MIPS n64 ABI
- A full range of branch and control operations, including conditional branches, conditional traps, jump-and-link, and system calls
- 4K L1 instruction and data caches which are direct-mapped, write-through, virtually indexed-physically tagged with 8-byte lines

- 16K shared L2 cache which is direct-mapped, write-back, physically indexed and tagged with 32-byte lines
- 64-bit integer ALU, including support for multiply and divide
- Coprocessor 0 (CP0), system control features such as a sufficiently capable MMU to implement OS virtual memory and paging features
- Multiple CPU protection rings (kernel, supervisor, usermode)
- Mature exception handling, including cycle timer, various arithmetic and memory access exceptions, and interrupt delivery from external devices
- Programmable interrupt controller (PIC) able to multiplex larger number of interrupt sources to the smaller number of IRQ lines supported by the MIPS ISA; this will also provide future support for inter-processor interrupts (IPIs)

At this time, the CHERI prototype omits a number of features found in the MIPS 4000, largely because they are not required to validate the research hypotheses we are exploring. Some other modifications were made due to the specific implementation characteristics of FPGA soft cores – in particular, the decision to implement smaller caches was motivated by the performance trade-offs in the FPGA substrate which provides comparatively high-speed main memory, as well as a desire for simplicity. The following features are omitted, or significantly modified, from the MIPS 4000 ISA:

- No floating point unit (CP1)
- Only 64-bit addressing mode; no 32-bit addressing support
- Only big endian support; no variable-endian features
- Currently, CHERI is a single-core, single-threaded processor; we have in-progress prototypes adding multithreading and multiprocessing support

The following sections provide more detailed information on the 64-bit MIPS subset implemented in the CHERI prototype.

2.1.1 MIPS instructions

CHERI implements roughly the instruction set found in the MIPS 4000, subject to high-level variations described in the previous section. The following tables document in greater detail the MIPS ISA instructions implemented in the CHERI prototype, followed by notes any limitations to specific instructions:

Instruction	Description	Status
`LB`	Load byte	Implemented
`LBU`	Load byte unsigned	Implemented
`LD`	Load doubleword	Implemented
`LDL`	Load doubleword left	Implemented
`LDR`	Load doubleword right	Implemented
`LH`	Load halfword	Implemented
`LHU`	Load halfword unsigned	Implemented
`LL`	Load linked	Implemented
`LLD`	Load linked doubleword	Implemented
`LUI`	Load upper immediate	Implemented
`LW`	Load word	Implemented
`LWL`	Load word left	Implemented
`LWR`	Load word right	Implemented
`LWU`	Load word unsigned	Implemented
`SB`	Store byte	Implemented
`SC`	Store conditional	Implemented
`SCD`	Store conditional doubleword	Implemented
`SD`	Store doubleword	Implemented
`SDL`	Store doubleword left	Implemented
`SDR`	Store doubleword right	Implemented
`SH`	Store halfword	Implemented
`SW`	Store word	Implemented
`SWL`	Store word left	Implemented
`SWR`	Store word right	Implemented

Table 2.1: MIPS load and store instructions in the CHERI prototype

Table	Description
Table 2.1	MIPS load and store instructions
Table 2.2	MIPS arithmetic instructions
Table 2.3	MIPS logical and bitwise instructions
Table 2.4	MIPS jump and branch instructions
Table 2.5	MIPS coprocessor instructions
Table 2.6	MIPS special instructions
Table 2.7	Additional instructions from other MIPS ISA versions

Instruction	Description	Status
`ADD`	Add	Implemented
`ADDI`	Add immediate	Implemented
`ADDIU`	Add immediate unsigned	Implemented
`ADDU`	Add unsigned	Implemented
`ADDI`	And immediate	Implemented
`DADD`	Doubleword add	Implemented
`DADDI`	Doubleword immediate	Implemented
`DADDIU`	Doubleword add immediate unsigned	Implemented
`DADDU`	Doubleword add unsigned	Implemented
`DDIV`	Doubleword divide	Implemented
`DDIVU`	Doubleword divide unsigned	Implemented
`DIV`	Divide	Implemented
`DIVU`	Divide unsigned	Implemented
`DMULT`	Doubleword multiple	Implemented
`DMULTU`	Doubleword multiple unsigned	Implemented
`DSUB`	Doubleword subtract	Implemented
`DSUBU`	Doubleword subtract unsigned	Implemented
`MFHI`	Move from **HI**	Implemented
`MFLO`	Move from **LO**	Implemented
`MTHI`	Move to **HI**	Implemented
`MTLO`	Move to **LO**	Implemented
`MULT`	Multiply	Implemented
`MULTU`	Multiply unsigned	Implemented
`SLT`	Set on less than	Implemented
`SLTI`	Set on less than immediate	Implemented
`SLTIU`	Set on less than immediate unsigned	Implemented
`SLTU`	Set on less than unsigned	Implemented
`SUB`	Subtract	Implemented
`SUBU`	Subtract unsigned	implemented

Table 2.2: MIPS arithmetic instructions in the CHERI ISA

Instruction	Description	Status
`AND`	And	Implemented
`DSLL`	Doubleword shift left logical	Implemented
`DSLLV`	Doubleword shift left logical variable	Implemented
`DSLL32`	Doubleword shift left logical + 32	Implemented
`DSRA`	Doubleword shift right arithmetic	Implemented
`DSRAV`	Doubleword shift right arithmetic variable	Implemented
`DSRA32`	Doubleword shift right arithmetic + 32	Implemented
`DSRL`	Doubleword shift right logical	Implemented
`DSRLV`	Doubleword shift right logical variable	Implemented
`DSRL32`	Doubleword shift right logical + 32	Implemented
`NOR`	Nor	Implemented
`OR`	Or	Implemented
`ORI`	Or immediate	Implemented
`SLL`	Shift left logical	Implemented
`SLLV`	Shift left logical variable	Implemented
`SRA`	Shift right arithmetic	Implemented
`SRAV`	Shift right arithmetic variable	Implemented
`SRL`	Shift right logical	Implemented
`SRLV`	Shift right logical variable	Implemented
`XOR`	Exclusive or	Implemented
`XORI`	Exclusive or immediate	Implemented

Table 2.3: MIPS logical and bitwise instructions in the CHERI ISA

Instruction	Description	Status
`BEQ`	Branch on equal	Implemented
`BEQL`	Branch on equal Likely	Implemented
`BGEZ`	Branch on greater than or equal to zero	Implemented
`BGEZAL`	Branch on greater than or equal to zero and link	Implemented
`BGEZALL`	Branch on greater than or equal to zero and link likely	Implemented
`BGEZL`	Branch on greater than or equal to zero likely	Implemented
`BGTZ`	Branch on greater than zero	Implemented
`BGTZL`	Branch on greater than zero likely	Implemented
`BLEZ`	Branch on less than or equal to zero	Implemented
`BLEZL`	Branch on less than or equal to zero likely	Implemented
`BLTZ`	Branch on less than zero	Implemented
`BLTZAL`	Branch on less than zero and link	Implemented
`BLTZALL`	Branch on less than zero and link likely	Implemented
`BLTZL`	Branch on less than zero likely	Implemented
`BNE`	Branch on not equal	Implemented
`BNEL`	Branch on not equal likely	Implemented
`J`	Jump	Implemented
`JAL`	Jump and link	Implemented
`JALR`	Jump and link register	Implemented
`JR`	Jump register	Implemented

Table 2.4: MIPS jump and branch instructions in the CHERI ISA

Instruction	Description	Status
`BCzF`	Branch on Coprocessor z False	Not implemented
`BCzFL`	Branch on Coprocessor z False Likely	Not implemented
`BCzT`	Branch On Coprocessor z True	Not implemented
`BCzTL`	Branch On Coprocessor z True Likely	Not implemented
`CFCz`	Move control from coprocessor	Not implemented
`COPz`	Coprocessor operation	See Section 2.1.4
`CTCz`	Move control to coprocessor	Not implemented
`DMFC0`	Doubleword move from system control coprocessor	See Section 2.1.2
`DMTC0`	Doubleword move to system control coprocessor	See Section 2.1.2
`LDCz`	Load doubleword to coprocessor	Not implemented
`LWCz`	Load word to coprocessor	Not implemented
`MFC0`	Move from system control coprocessor	See Section 2.1.2
`MFCz`	Move from coprocessor	See Section 2.1.4
`MTC0`	Move to system control coprocessor	See Section 2.1.2
`MTCz`	Move to coprocessor	See Section 2.1.4
`SDCz`	Store doubleword from coprocessor	Not implemented
`SWCz`	Store word from coprocessor	Not implemented
`TLBP`	Probe TLB for matching entry	Implemented
`TLBR`	Read indexed TLB entry	Implemented
`TLBWI`	Write indexed TLB entry	Implemented
`TLBWR`	Write random TLB entry	Implemented

Table 2.5: MIPS coprocessor instructions in the CHERI ISA; see Section 2.1.2 for limitations on CP0 instructions; see Section 2.1.4 for limitations on generic coprocessor instructions

Instruction	Description	Status
`BREAK`	Breakpoint	Implemented
`CACHE`	Cache	Implemented
`ERET`	Exception return	Implemented
`SYNC`	Synchronise	See Section 2.1.5
`SYSCALL`	System call	Implemented
`TEQ`	Trap if equal	Implemented
`TEQI`	Trap if equal immediate	Implemented
`TGE`	Trap if greater than or equal	Implemented
`TGEI`	Trap if greater than or equal immediate	Implemented
`TGEIU`	Trap if greater than or equal immediate unsigned	Implemented
`TGEU`	Trap if greater than or equal unsigned	Implemented
`TLT`	Trap if less than	Implemented
`TLTI`	Trap if less than immediate	Implemented
`TLTIU`	Trap if less than immediate unsigned	Implemented
`TLTU`	Trap if less than unsigned	Implemented
`TNE`	Trap if not equal	Implemented
`TNEI`	Trap if not equal immediate	Implemented

Table 2.6: MIPS special instructions in the CHERI ISA

Instruction	Description	Version	Status
`MADD`	Multiply and add unsigned words to **HI**, **LO**	MIPS32	Implemented
`MOVN`	Move conditional on not zero	MIPS32	Implemented
`MOVZ`	Move conditional on zero	MIPS32	Implemented
`MSUB`	Multiply and subtract word to **HI**, **LO**	MIPS32	Implemented
`MUL`	Multiply word to general-purpose register	MIPS32	*Untested*
`SSNOP`	Superscalar no operation	MIPS32	See Section 2.1.5
`WAIT`	Enter standby mode	MIPS32	Not implemented

Table 2.7: Selected additional instructions in the CHERI ISA, derived from other MIPS ISA versions; see Section 2.1.5

2.1.2 Limitations to coprocessor 0 support

Currently, not all system control coprocessor (CP0) features are implemented in the CHERI prototype. In the future, this section will document included CP0 features explicitly.

2.1.3 Modifications to the MIPS TLB model

The MIPS 4000 MMU implements a 48-entry, fully associative Translation Look-aside Buffer (TLB). Software interacts with the MIPS 4000 TLB by performing Write Indexed or Write Random operations where Write Random operations use the Random CP0 register as the index. The Random CP0 register decrements every cycle but resets to the highest TLB entry when it reaches the Wired CP0 register. Thus, a Write Random operation never overwrites TLB entries below the Wired register.

CHERI's TLB is composed of a lower 8 fully associative entries and an upper 32 direct mapped entries. This is due to our desire to have large TLB but the inability of FPGA fabrics to construct large associative searches efficiently. This associative plus mapped structure allows an arbitrarily large TLB with trivial additional logic because most of the entries are stored in block ram.

Our implementation behaves differently from a simple MIPS 4000 TLB in several ways:

1. Writing arbitrary indexed values is supported only for the lower 8 entries and an indexed write to the upper 32 entries will result in a write to an index above 7 whose lower 5 bits are equal to the lower 5 bits of the virtual page number.

2. Write Random operations will not write to a random location but will also write to an index above 7 whose lower 5 bits are equal to the lower 5 bits of the virtual page number of the written entry.

3. A valid entry that is displaced by a write random instruction will be placed in an unpredictable location above the wired entry and less than or equal to 8. Thus, the fully associative entries that are not wired act as a victim buffer for the direct mapped entries.

4. The MIPS 4000 supports variably-sized pages, but CHERI currently supports fixed 4-kilobyte pages.

5. CHERI's TLB implementation also does not support 32-bit virtual addresses as MIPS 4000 does.

6. CHERI's TLB supports a full 64-bit virtual address space rather than MIPS 4000 40-bit virtual addresses.

CHERI's TLB implementation works for FreeBSD without modification. FreeBSD wires only the bottom TLB entry for use in exception handlers and then accesses the rest of the TLB entries chiefly using the Write Random operation with Write Indexed operations only being used to modify entries in place or to invalidate TLB entries. When FreeBSD invalidates TLB entries, it uses a virtual page number whose lower bits are equal to the index number in the TLB, so our design which takes the lower bits of the virtual page number as the index works as expected. Furthermore, FreeBSD always probes to find the index of an entry immediately before modifying it and does not maintain a memory of where it has placed entries in the table. Thus, FreeBSD is not confused when our implementation relocates entries from the direct mapped region to the victim buffer.

If any operating system or hypervisor desired to more closely manage the TLB, it should take into account the mapped nature of the upper entries of the TLB and the possibility that a Write Random operation may relocate a previously mapped entry.

In other respects, the CHERI TLB is similar to the MIPS 4000, including each entry mapping two pages and 36-bit physical addresses.

2.1.4 Limitations to generic coprocessor support

The CHERI prototype does does currently support generic coprocessor instructions, but does implement an interface for the capability coprocessor using coprocessor 2.

2.1.5 Selected ISA additions from later MIPS versions

Table 2.7 documents selected instructions added to the CHERI ISA from later MIPS versions. In general, we have added instructions only where required by common compiler toolchains and operating systems.

As the CHERI prototype is pipelined, but currently neither superscalar nor multicore, the `SSNOP` and `SYNC` instructions are interpreted as a `NOP`s. If superscalar support is added to future CHERI versions, it will need to be enhanced.

The CHERI prototype also implements the `config1` CP0 shadow register, introduced in MIPS32, which allows querying cache layout properties. This is used by FreeBSD during CPU discovery to select cache management routines.

2.1.6 Virtual address space

The 64-bit MIPS ISA divides its 64-bit address space into a number of segments with various properties. CHERI implements roughly the same address space layout as the MIPS r4000, except that CP0 status register bits **UX**, **SX**, and **KX** are always set to 1. This means that user mode, supervisor mode, and kernel mode must always execute in 64-bit addressing mode in CHERI (i.e., no 32-bit addressing is supported). In addition, the mapped regions xsuseg and xsseg have been extended 62 bits instead of stopping at 40 bits as in the MIPS r4000. This matches the more recent MIPS64 spec.

Ring 2: user mode

Table 2.8 illustrates the user mode address space. The processor is in user mode whenever **KSU** is 10, **EXL** is 0, and **ERL** is 0. In this mode, only the lower quarter of the address space is available and all addresses are virtual and mapped by the TLB.

Start address	Stop address	Description	Size
`0x40000000 00000000`	`0xffffffff ffffffff`	address error	
`0x00000000 00000000`	`0x3fffffff ffffffff`	xuseg (user)	2^{62} Bytes

Table 2.8: CHERI address space layout in user mode

Start address	Stop address	Description	Size
`0xffffffff e0000000`	`0xffffffff ffffffff`	address error	
`0xffffffff c0000000`	`0xffffffff dfffffff`	csseg	512M
`0x80000000 00000000`	`0xffffffff bfffffff`	address error	
`0x40000000 00000000`	`0x7fffffff ffffffff`	xsseg	2^{62} Bytes
`0x00000000 00000000`	`0x3fffffff ffffffff`	xsuseg (user)	2^{62} Bytes

Table 2.9: CHERI address space layout in supervisor mode

Ring 1: supervisor mode

Table 2.9 illustrates the supervisor address space. The processor is in supervisor mode whenever **KSU** is 01, **EXL** is 0, and **ERL** is 0. All available addresses are virtual and mapped by the TLB.

Ring 0: kernel mode

Table 2.10 illustrates the kernel address space; Table 2.11 details the xkphys subset of the address space, in which the physical memory space is mapped directly into regions of virtual address space, using various caching policies. The processor is in kernel mode if **KSU** is 0, **EXL** is 1, or **ERL** is 1.

Start address	Stop address	Description	Size
`0xffffffff e0000000`	`0xffffffff ffffffff`	ckseg3 - mapped	512M
`0xffffffff c0000000`	`0xffffffff dfffffff`	cksseg - mapped	512M
`0xffffffff a0000000`	`0xffffffff bfffffff`	ckseg1 - unmapped, uncached	512M
`0xffffffff 80000000`	`0xffffffff 9fffffff`	ckseg0 - unmapped, cached	512M
`0xc00000ff 80000000`	`0xffffffff 7fffffff`	address error	
`0xc0000000 00000000`	`0xc00000ff 7fffffff`	xkseg - mapped	
`0x80000000 00000000`	`0xbfffffff ffffffff`	xkphys - unmapped	
`0x40000000 00000000`	`0x7fffffff ffffffff`	xsseg	2^{62} Bytes
`0x00000000 00000000`	`0x3fffffff ffffffff`	xsuseg (user)	2^{62} Bytes

Table 2.10: CHERI address space layout in kernel mode

Start address	Stop address	Description
`0xb8000000 00000000`	`0xbfffffff ffffffff`	reserved
`0xb0000000 00000000`	`0xb7ffffff ffffffff`	cached, coherent update on write
`0xa8000000 00000000`	`0xafffffff ffffffff`	cached, coherent exclusive on write
`0xa0000000 00000000`	`0xa7ffffff ffffffff`	cached, coherent exclusive
`0x98000000 00000000`	`0x9fffffff ffffffff`	cached, noncoherent
`0x90000000 00000000`	`0x97ffffff ffffffff`	uncached
`0x88000000 00000000`	`0x8fffffff ffffffff`	reserved
`0x80000000 00000000`	`0x87ffffff ffffffff`	reserved

Table 2.11: Layout of the xkphys region in CHERI

Instruction	Description	Status
`CGetBase`	Move base to a general-purpose register	Implemented
`CGetLen`	Move length to a general-purpose register	Implemented
`CGetPerm`	Move permissions field to a general-purpose register	Implemented
`CGetType`	Move object type field to a general-purpose register	Implemented
`CGetUnsealed`	Move unsealed flag to a general-purpose register	Implemented
`CGetPCC`	Move the PCC and PC to a general-purpose registers	Implemented
`CGetTag`	Move the valid capability flag to a general-purpose register	Implemented
`CSetType`	Set the object type field of an executable capability	Implemented
`CIncBase`	Increase Base	Implemented
`CSetLen`	Decrease Length	Implemented
`CAndPerm`	Restrict Permissions	Implemented
`CClearTag`	Clear the capability valid flag	Implemented

Table 2.12: CHERI ISA instructions for getting and setting capability register fields

2.2 CHERI ISA extensions

The CHERI prototype currently has an initial implementation of the CHERI ISA capability extensions. Testing of capability instructions has thus far been limited to high-level ISA properties, with only limited testing of certain features (such as exception generation and priority). However, many basic features of memory capabilities (segments) are usable, and have been demonstrated as part of CheriBSD and the Deimos microkernel, described in the *CHERI User's Guide*. Tagged memory has now also been implemented, experimentally.

The following tables document in greater detail the CHERI ISA instructions implemented in the CHERI prototype, followed by notes any limitations to specific instructions:

Table	Description
Table 2.12	Getting and setting capability fields
Table 2.13	Loading and storing [via] capabilities
Table 2.14	Instructions relating to object capabilities

2.3 CHERI2 Specific Features

In general CHERI2 implements the same MIPS subset and capability extensions as documented in the previous sections. However, it also implements some other experimental features, and there are some minor differences as described here. **N.B. CHERI2 is a rapidly developing prototype, therefore it is wise to check with the developers before relying on any of this information.**

Instruction	Description	Status
`CSC`	Store Capability	Implemented
`CLC`	Load Capability	Implemented
`CLB`	Load Byte Via Capability Register	Implemented
`CLH`	Load Half-Word Via Capability Register	Implemented
`CLW`	Load Word Via Capability Register	Implemented
`CLD`	Load Double Via Capability Register	Implemented
`CLBU`	Load Byte Unsigned via Capability Register	Implemented
`CLHU`	Load Half-Word Unsigned via Capability Register	Implemented
`CLWU`	Load Word Unsigned via Capability Register	Implemented
`CSB`	Store Byte Via Capability Register	Implemented
`CSH`	Store Half-Word Via Capability Register	Implemented
`CSW`	Store Word Via Capability Register	Implemented
`CSD`	Store Double Via Capability Register	Implemented
`CSBH`	Store High Byte Via Capability Register	Not tested
`CSHH`	Store High Half-Word Via Capability Register	Not tested
`CSWH`	Store High Word Via Capability Register	Not tested
`CLLD`	Load Linked Double Via Capability Register	Unimplemented
`CSCD`	Store Conditional Double Via Capability Register	Unimplemented

Table 2.13: CHERI ISA instructions for loading and storing [via] capabilities

Instruction	Description	Status
`CGetCause`	Read capability exception cause register	Implemented
`CJR`	Jump Capability Register	Implemented
`CJALR`	Jump and link Capability Register	Implemented
`BC2F`	Branch if a capability is valid	Unimplemented
`CSealCode`	Seal an executable capability	Implemented
`CSealData`	Seal a non-executable capability with the object type of an executable capability	Implemented
`CUnseal`	Unseal a sealed capability	Implemented
`CCall`	Protected procedure call into a new security domain	Unimplemented
`CReturn`	Return to the previous security domain	Unimplemented

Table 2.14: CHERI ISA instructions for creating and invoking object capabilities

2.3.1 Multi-Threading

CHERI2 implements a simple form of multi-threading. It supports a configurable, statically determined number of hardware threads which are rotated on a cycle-by-cycle basis (barrel scheduled). In the current implementation all threads are runnable at all times, so there is no way to suspend a thread in hardware.

To build with multiple threads, edit the `ThreadSZ` definition in `cheri2/trunk/CHERITypes.bsv`. This sets the number of bits of thread ID, which determines the number of threads, $T = 2^{\texttt{ThreadSZ}}$. By default this is set to 0, for single-threaded execution.

Most supported CP0 registers are maintained independently for each thread, with the following notable exceptions:

PrID (15) Read-only, bits 31:24 contain current thread ID.

COUNT (9) Read-only and consistent between threads. Each thread has its own COMPARE register.

CAUSE:IP6..2 (13) External hardware interrupts are delivered to all threads, however the built-in timer interrupt (IP7) and software interrupt bits (IP0 and IP1) are maintained per thread. Each thread can control the delivery of interrupts individually via the mask field in the status register.

TLB The TLB is currently shared between all threads. Although the relevant CP0 registers are replicated per thread, it is still necessary for software to perform synchronisation around TLB operations in order to avoid race conditions e.g. between `PROBE` and `TLBR` or `TLBWI` instructions.

Chapter 3

The CHERI Programmable Interrupt Controller

This chapter describes the Programmable Interrupt Controller (PIC) attached to each CHERI core. The PIC provides a simple way for mapping a potentially large number of external interrupts onto the small set of hardware interrupts defined by the MIPS ISA (see Table 3.3). On CHERI2 each interrupt must also be mapped to a particular hardware thread. The PIC exposes memory mapped registers which can be used by software to configure the mapping and also to set, clear and read pending interrupts. Thus the PIC allows interrupts to be triggered by both hardware-wired peripherals (e.g., a UART) and by software, referred to respectively as 'hard' and 'soft' sources. This latter facility can be used for inter-processor interrupts (IPIs) on multi-threaded and multicore configurations.

3.1 Sources

The PIC consists of S sources, which may be either hard or soft.

Soft The value of a soft source comes from its interrupt pending (IP) state bit, which can be set or cleared by software. In the future it may also be possible for these to be set by some external event, such as a message received over the inter-core interconnect.

Hard The value of a hard source comes directly from a peripheral device, and is not latched. Hard sources also have an IP bit which may be manipulated in the same way as for soft sources, but this is mainly for debugging purposes.

To calculate the current state for a particular MIPS interrupt the PIC ORs the value of all sources which are enabled and mapped to that interrupt.

3.2 Source Numbers and Base Addresses on CHERI

The source numbers and register base addresses are as shown in Tables 3.1 and 3.2.

Source No	Use
0–63	Up to 64, hard-wired external interrupts
64–127	64 implemented soft interrupts
128–1023	Reserved soft interrupts, unimplemented

Table 3.1: CHERI PIC source number allocation

Register Name	Address	Used to
PIC_CONFIG_BASE	Refer to relevant section of Chapter 9	Configure interrupts source mappings
PIC_IP_READ_BASE	$\texttt{PIC\_CONFIG\_BASE} + 8 * 1024$	Read interrupt source state
PIC_IP_SET_BASE	$\texttt{PIC\_IP\_READ\_BASE} + 128$	Set interrupt pending bits
PIC_IP_CLEAR_BASE	$\texttt{PIC\_IP\_READ\_BASE} + 256$	Clear interrupts pendings btis

Table 3.2: CHERI PIC control register addresses

3.3 Config Registers: PIC_CONFIG_X

Each source has an associated configuration register with the format shown in Figure 3.3. This register allows the interrupt to be directed to a given interrupt number of a given thread and to enable/disable delivery of the interrupt. The configuration register for source s has address $\texttt{PIC\_CONFIG\_BASE} + 8s$. The default value for all configuration fields is 0 (i.e. disabled). Word or double word accesses may be used.

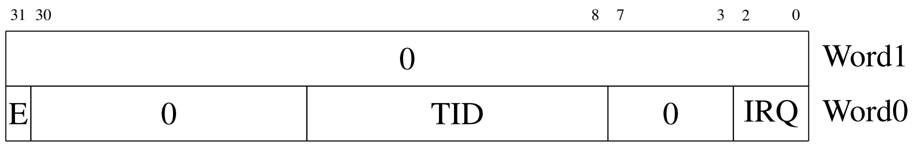
3.4 Interrupt Pending Bits

Each interrupt source has 1-bit of interrupt pending (IP) state associated with it.

For hardware sources the IP bit is ORed with the incoming interrupt wire to provide the current value for the interrupt source. This may be used to artificially trigger an interrupt for debugging purposes. Note that the hardware interrupt is *not* latched by the IP bit, therefore the source will stay high only as long as the hardware source asserts its interrupt, and will go low once software has dealt with the interrupt at the device. This behaviour is consistent with the IP bits in the MIPS cause register.

The PIC also provides soft sources which may be used for inter-thread interrupts. It is expected that software will configure at least one soft source per thread to be used for this. If non-maskable or debug inter-thread interrupts are also required then two or more sources per thead may be configured. On future multi-processor builds it will be possible for an interrupt pending bit to be set by a message on the inter-processor interconnect, thus allowing for inter-processor interrupts, or message based interrupts similar to PCI's Message Signalled Interrupts.

The IP bits are packed into 64-bit registers for manipulation by software. The current value for a



0	R/W	Zero reserved: zero on read, should be written with zero.
E	R/W	Enable/disable this interrupt source. If set to one the interrupt will be enabled, otherwise it will be masked.
TID	R/W	Thread ID of the hardware thread which will receive this interrupt. The width of this depends on the number of hardware threads implemented on the core.
IRQ	R/W	MIPS interrupt number to which this interrupt source will be delivered. Values 0-7 are mapped to the MIPS interrupts as shown in Table 3.3.

Figure 3.1: PIC Configuration Register Format

IRQ Field Value	MIPS Interrupt
0-4	MIPS external interrupts 0-4 corresponding to IP2-IP6 in the CP0 Cause register.
5	MIPS external interrupt 5 corresponding to IP7 in the CP0 Cause register. The value of IP7 is the logical OR of the output of the PIC with the core's timer interrupt value.
6	MIPS non-maskable interrupt corresponding to NMI field of the CP0 Status register. CURRENTLY NOT SUPPORTED ON CHERI OR CHERI2.
7	MIPS EJTAG debug exception trigger. CURRENTLY NOT SUPPORTED ON CHERI OR CHERI2.

Table 3.3: Values of the IRQ field of config register.

source, s, can be read from a read-only register at address `PIC_IP_READ_BASE`$+\lfloor s/8 \rfloor$, in bit $s \bmod 64$ (numbered from zero as the least significant bit). For hard sources this value is the value of the external interrupt wire ORed with the IP bit, thus it is not possible to read the state of the IP bit in isolation. Software may set the IP bit for a source by writing a one to the corresponding offset from `PIC_IP_SET_BASE` and, similarly, clear it using an offset from `PIC_IP_CLEAR_BASE`. Bits written with zero will have no effect. The set/clear semantics allows for atomically manipulating one or more bits in the packed registers without the potential for races associated with a read/modify/write sequence.

3.5 Reset State

On reset all PIC configuration and state is set to zero except for the first five hardware sources, 0-4, which are given backwards compatible initial state as follows:

- The E bit is set to one to enable the interrupt
- The TID field is set to zero to pass the interrupt to thread 0
- The IRQ field is set to the source number

The effect of this is to make the PIC completely transparent to PIC unaware code, which may behave as if external interrupts were directly connected to the MIPS interrupt wires. PIC aware software should not rely on this behaviour and should configure all interrupt sources explicitly on boot.

3.6 Safe handling of interrupts

It is expected that the soft interrupts will be used in combination with shared-memory communication to pass inter-thread or inter-processor messages. In order to do this safely without any missed wakeups the source should be cleared first, before handling any incoming messages in a loop. This could result in a spurious interrupt in the case where a second interrupt arrives whilst processing the first interrupt, but will not result in missed wakeups.

For hardware sources it is assumed that the device will provide a safe way of handling and quiescing the interrupt.

Chapter 4

The CHERI processor implementation

This chapter provides a high-level overview of the CHERI prototype processor implementation, including programming language choice, directory layout, and a high-level description of the prototype's files and modules.

4.1 Bluespec

The CHERI prototype is implemented in the Bluespec Hardware Description Language (HDL), a Haskell-derived programming language that allows highly parameterised and structured logic designs. Bluespec source code may be compiled to an efficient C simulation, or into Verilog for simulation or synthesis. One of the key properties of Bluespec is that it makes design space exploration far more accessible than traditional low-level HDLs, which is critical to allowing us to evaluate quickly and easily the impact of our design choices on a practical hardware implementation.

4.2 Directory layout

The Bluespec source code for the CHERI processor resides in the root of the CHERI distribution. A series of sub-directories, listed in Table 4.1, contain a combination of supplementary software source code including an interactive self-test and unit test suite, tools used in building CHERI, and are targets for generated files.

4.3 Key files

Table 4.2 describes the key files in the CHERI prototype implementation.

Directory	Description
`trunk/`	Root of the CHERI source tree, home of Bluespec source code
`boards/`	Holds project directories various for FPGA boards
`bsim/`	Destination for generated Bluespec simulator files
`sw/`	Home of integrated software component source code
`verilog/`	Destination for generated Verilog files
`vsim/`	Destination for generated Verilog simulator files

Table 4.1: Directories in the CHERI source code distribution

4.4 Conditionally compiled features

Table 4.3 is a list of conditionally compiled features in the CHERI processor. Values for these macros are selected by various build targets described in Chapter 5, and passed to the Bluespec compiler via the `bsc` command line.

File	Description
`DE4_MIPS.qar`	Altera Quartus archive for CHERI on the Terasic DE4 configuration, if present
`makefile`	GNU makefile to build CHERI
`setup.sh`	Configure Cambridge Bluespec build environment
`MIPS.bsv`	Types and shared functions for the design
`MIPSTop.bsv`	Top-level module implementing instruction and register fetch, which instantiates all other modules
`Decode.bsv`	Decode stage of the pipeline
`Execute.bsv`	Execute stage of the pipeline
`MemAccess.bsv`	Memory access and writeback stages of the pipeline
`Memory.bsv`	Memory subsystem, which instantiates the caches, merging logic, and memory interface
`Cache.bsv`	4KB L1 Cache for both instruction and data caches
`TopAvalon256Merge.bsv`	Module merging all memory interfaces into a single interface to feed the L2 cache
`L2Cache.bsv`	16K L2 cache
`TopAvalonPhy.bsv`	Top-level module adapting CHERI's memory interface to an Avalon bus interface
`TopSimulation.bsv`	Top-level module interfacing CHERI's memory interface with a DRAM simulation and the PISM bus for C peripheral models
`BRAM1ServerBE256.bsv`	Module for single port, 256-bit wide, byte enabled BRAMs, needed to simulate DRAM in TopSimulation.bsv
`MIPSRegFile.bsv`	Three-port register file
`AltBRAMCore.bsv`	Generic BRAM package that directly instantiates Altera BRAMs
`CP0.bsv`	Coprocessor 0 containing all configuration registers
`TLB.bsv`	40-entry TLB with three cached interfaces
`Scheduler.bsv`	Context after a pipeline flush
`CapCop.bsv`	Module implementing the capability coprocessor
`getChar.c`	C function for simulating a UART

Table 4.2: Key files in the CHERI source code

Macro	Description
`BLUESIM`	Compile for Bluespec simulation rather than FPGA target
`DEBUG`	Enable hardware-level tracing
`DEBUG_INST`	Enable ISA extensions to dump CPU state in simulation
`TRACE`	Enable instruction-level tracing
`CAP`	Include capability coprocessor

Table 4.3: Macros controlling conditionally compiled features in CHERI

Chapter 5

Simulating CHERI

This chapter describes how to check out the CHERI source code, build the CHERI simulator, and run the CHERI unit test suite. It describes various build targets for the simulator with varying levels of tracing. This documentation assumes access to the following resources:

- 32-bit or 64-bit Ubuntu Linux 10.04.2 LTS or 12.04 LTS workstation or server
- Access to the hosted Subversion repository

5.1 Software dependencies to build CHERI

Table 5.1 documents software packages required to build the CHERI simulator and test suite. For Altera tools to function under Ubuntu, `/bin/sh` must point to bash, and not dash, which is the default in Ubuntu. This can be accomplished with the following commands:

```
# rm /bin/sh
# ln -s /bin/sh bash
```

To install an Ubuntu package such as those listed in the table, using the following command:

```
# apt-get install <package-name>
```

If Ubuntu 10.04 (Lucid) or 9.10 (Karmic) is used, the backports source must be enabled in `/etc/apt/sources.list` to install libstdc++5.

The SDE cross-compiling distribution of the GCC compiler on Linux should be available from MIPS.com. The current URL is unknown.

The Cambridge extended assembler is available from `/usr/groups/ctsrd` on the network filer at the Cambridge Computer Laboratory.

Program	Ubuntu Package	Required to
Subversion	subversion	Check out
GNU make	make	Build, run test suite
GNU Bison 2.4.1	bison	Build simulator
Flex 2.5.35	flex	Build simulator
Bluespec 2011.05.beta1	See Bluespec Documentation	Build
GNU C++ libraries (v5)	libstdc++5	Running Bluespec
Perl	perl	Build
SDL 1.2.14	libsdl1.2-dev libsdl1.2debian	Build, run simulated tPad frame buffer
Python 2.6	python	Build, run test suite
SDE toolchain 6.06	none	Build test suite
Cambridge extended assembler	none	Build test suite
Nose 0.11.1	python-nose	Run test suite
Ubuntu 32-bit libraries	ia32-libs	Build test suite on 64-bit systems

Table 5.1: Software build dependencies for CHERI components

5.2 Checking out CHERI

The Cambridge Subversion repository uses SSH authentication keys as capabilities to identify what repository and what rights are held by a client. By default, SSH will offer keys held by the agent (or in your home directory) in the order it finds them, which, if you hold multiple keys to different repositories on the Subversion server, may lead SSH to select the wrong key. It is therefore necessary to ensure that the right SSH key is used; one way to do so is to create a new SSH agent, adding only the appropriate key to that session[1]. To set up an SSH agent in this manner, use something like the following:

```
$ ssh-agent bash
$ ssh-add ~/.ssh/id_ctsrd_rsa
```

It is also possible to configure `.ssh/config` to offer only a specific key to specific servers – see the SSH man page for details. To perform an initial checkout of CHERI, use the following Subversion command:

```
$ svn co svn+ssh://secsvn@svn-ctsrd.cl.cam.ac.uk/ctsrd ctsrd
```

To update an existing checkout of CHERI, use the following Subversion command:

```
$ cd ctsrd
$ svn update
```

5.3 Building the CHERI simulator

The CHERI source code and build tools may be found in the `ctsrd/cheri/trunk` directory tree. Before building CHERI, you must configure the Bluespec development environment:

```
$ cd ctsrd/cheri/trunk
$ source setup.sh
```

The CHERI build supports a number of targets depending on whether the goal is to build a Bluespec simulator or Verilog for simulation or synthesis and whether the capability coprocessor is to be included. Table 5.2 lists high-level build targets for CHERI. A typical run of the simulator might be initiated using:

```
$ make sim_cap
$ ./sim
```

The CHERI build is also sensitive to two make variables. If `WITH_UNPIPELINE` is set, CHERI will build to single-step the pipeline every 16 cycles by default. If `PAUSE` is set, CHERI will begin

[1]On Mac OS X, new `ssh-agent` sessions inherit all SSH keys added to the user keychain, so you must run `ssh-add -D` to flush them. This is not required on other platforms

Target	CP2
`sim`	No
`sim_cap`	Yes

Table 5.2: CHERI simulator build targets

paused by default and will need to be prompted to continue by a command to the debug unit, whose interface is described in Chapter 6.

The CHERI executable is sensitive to four arguments. The `+trace` argument will give a concise report for each instruction committed. The `+cTrace` argument will report the number of dead cycles between committed instructions. The `+regDump` argument will enable the debug instructions which report the contents of the register files. The `+debug` argument will report all debug output from the internal processor state.

The build also compiles an interactive software test tool, found in the `sw` sub-directory. Other available software packages, stored elsewhere in the repository, include the CHERI unit test suite described in Chapter 7 (`cheritest/trunk`) and the CHERI demonstration environment described in the *CHERI User's Guide* (`deimos/trunk`). When the simulator is run, it loads a memory image from the current working directory; running the simulator from the root of the CHERI development tree will automatically load the interactive test tool.

5.4 Configuring the CHERI Simulator

At startup, the simulator tries to read a file `./simconfig` or the file pointed to by the environment variable `CHERI_CONFIG`. This file describes in a C-like syntax how the hardware should be simulated. A valid configuration must exist or the simulator will not start. A default configuration file is included in the `cheri/trunk` directory.

Individual simulated hardware peripherals are built as shared libraries. The simulator will attempt to `dlopen()` these shared libraries as they are encountered in the configuration file. Any module-specific options are passed to the module at load-time. If a module fails to load, either because it cannot be found or because invalid options were given, the simulation will terminate with an assertion failure. Figure 5.1 illustrates a sample `simconfig` file.

First, a series of simulated device modules are loaded using `module` statements; paths must be specified. Then, a series of devices is declared using `device` blocks, which must each declare a `class`, which selects the simulated device type; for each device, at least a base address (`addr`) and length (`length`) must be specified. An optional `irq` can be set, as well as device class-specific parameters, such as socket types, file paths, and so on. Devices can be conditionally defined based on whether environmental variables have been set, with both a positive `ifdef` and negative `ifndef` syntaxes permitted. Finally, options can be set from environmental variables using the `getenv` syntax; if the variable is not set, than an empty string will be used for the value.

```
module ../../cherilibs/trunk/peripherals/dram.so
module ../../cherilibs/trunk/peripherals/ethercap.so
module ../../cherilibs/trunk/peripherals/uart.so

device "dram0" {
        class dram;
        addr 0x0;
        length 0x40000000;
};
ifdef "CHERI_KERNEL" device "kernel" {
        class dram;
        addr 0x100000;
        length 0xff00000;
        option path getenv "CHERI_KERNEL";
        option type "mmap";
        option cow "yes";
};
ifdef "CHERI_SDCARD" device "sdcard0" {
        class sdcard;
        addr 0x7f008000;
        length 0x400;
        option path getenv "CHERI_SDCARD";
        option readonly "yes";
};
ifndef "CHERI_CONSOLE_SOCKET" device "uart0" {
        class uart;
        addr 0x7f000000;
        length 0x20;
        irq 0;
        option type "stdio";
}
ifdef "CHERI_CONSOLE_SOCKET" device "uart0" {
        class uart;
        addr 0x7f000000;
        length 0x20;
        irq 0;
        option type "socket";
        option path getenv "CHERI_CONSOLE_SOCKET";
}
```

Figure 5.1: Example simconfig configuration file

5.5 Simulating CHERI

When the `sim` or `sim_cap` target is used, a simulator binary, `sim` is generated. The simulator automatically loads a physical memory image from a series of hex memory files, `mem0.hex`, `mem1.hex`, ... `mem7.hex`, used to populate initial memory contents for BRAM. By default, the memory image generated from `sw` contains a small interactive test suite that communicates via a simulated serial I/O hooked up to the simulator's standard input and output streams.

5.6 Running the CHERI test suite

The `cheritest/trunk` subtree contains a MIPS ISA unit test suite that exercises various processor features, including initial register values, memory access, jump instructions, exceptions, and so on. `make test` in the `cheri/trunk` tree will run the test suite; a detailed discussion of the test suite appears in Chapter 7.

Chapter 6

Using the CHERI debug unit

The CHERI prototype includes a simple debug unit that communicates with an external host using a two-way streaming 8-bit interface. The debug unit can pause and step the pipeline, set breakpoints, and insert instructions into the pipeline. Instructions inserted by the debug unit may use operands from the debug unit as well as operands from the register file. The result of an instruction from the debug unit may be written back to the debug unit as well as the register file. These debug unit elements may be used to implement a variety of higher-level services, including a proxy from the GDB server protocol, and an external memory image loader.

6.1 Communicating with the CHERI debug unit

The CHERI debug unit communicates with a host computer over a two-way streaming 8-bit interface. The current system uses an Altera JTAG streaming component which is tunnelled over USB to the host PC. The Altera System Console utility may send and receive bytes to and from the debug unit using a desktop computer connected with a USB cable. All commands and responses are defined as series of bytes sent or received over this streaming channel.

Commands and responses traveling over the channel are arranged as messages. Every message begins with two bytes. The first is the message type and the second is the message length. If the message length is non-zero, the prescribed number of bytes follow the two bytes of the message header.

6.2 CHERI debug registers

The CHERI debug unit has eight registers:

- Debug Instruction
- Operand A
- Operand B

Instruction	Command	Length	Payload
Load Instruction	i	4	Instruction
Load Operand A	a	8	64-bit Operand
Load Operand B	b	8	64-bit Operand
Execute Instruction	e	0	
Report Destination	d	0	
Load Breakpoint 0	0	8	64-bit Address
Load Breakpoint 1	1	8	64-bit Address
Load Breakpoint 2	2	8	64-bit Address
Load Breakpoint 3	3	8	64-bit Address
Pause Execution	p	0	
Resume Execution	r	0	
Step Execution	s	0	
Resume Execution	r	0	
Move PC to Destination	c	0	
Resume Unpipelined	u	0	

Table 6.1: Instruction messages for the debug unit

- Breakpoint 0
- Breakpoint 1
- Breakpoint 2
- Breakpoint 3
- Destination

The first seven registers can be written and the Destination register can be read.

6.3 CHERI debug instructions

The CHERI debug unit supports the instructions listed in Table 6.1. The "Command" is the ASCII character that should appear in the first byte of the message sent to the debug unit, that is, the message type. The "Length" is the value of the second byte of the message. The "Payload" is the contents of the following bytes, equal in number to the value of the "Length" field. All instructions will produce a response from the debug unit confirming completion of the request. These message types are listed in Section 6.4.

Notes on Some Instructions:

Load Instruction Load an instruction into the debug unit's instruction register in preparation to insert it into the processor pipeline.

Load Operand A & B Load values into the Operand A & B registers to be possibly used as operands of the instruction in the Instruction register.

Execute Instruction Insert the instruction contained in the Instruction register into the processor pipeline.

Report Destination Report the 64-bit (8-byte) value in the Destination register. The debug unit will send a message containing the contents of the destination register back to the debugger.

Load Breakpoint 0-3 Load an address into one of the breakpoint registers and arm that breakpoint. When the next program counter is equal to one of the breakpoint registers, the processor will automatically pause and the value of the breakpoint, which when fired will be sent in a message to the debugger. Loading the address `0xffffffffffffffff` will disable a breakpoint.

Step Execution Step one instruction if the processor is paused. If the next instruction is a branch, the branch delay slot and the branch target will also be executed.

Accessing Debug Registers from Debug Instructions When instructions originate from the debug unit, references to **R0** are interpreted as references to registers in the debug unit. An instruction from the debug unit which takes two operands from **R0** and writes back to **R0** will take Operand A and Operand B and will write back to the Destination register in the debug unit. In general, if "rs" refers to **R0**, that operand will come from Operand A in the debug unit and if "rt" refers to **R0**, that operand will come from Operand B in the debug unit.

6.4 CHERI debug responses

Table 6.2 lists message types that the debug unit may generate. All of them are direct responses to instructions except for the "Breakpoint Fired" command which might be delivered at any time.

Notes for Some Responses:

Execute Instruction and Exception Responses When an "Execute Instruction" command is received, the debug unit will return an "Execute Instruction Response" message if execution of the instruction did not throw an exception. If the instruction generated an exception, the debug unit will return an "Execute Exception Response" message with a payload of one byte which will contain the 4-bit MIPS exception code generated by the instruction.

Message	Type Byte	Length	Payload
Load Instruction Response	`0xe9`	0	
Load Operand A Response	`0xe1`	0	
Load Operand B Response	`0xe2`	0	
Execute Instruction Response	`0xe5`	0	
Execute Exception Response	`0xc5`	1	MIPS Exception Code
Report Destination Response	`0xe4`	8	64-bit Value
Load Breakpoint 0 Response	`0xb0`	0	
Load Breakpoint 1 Response	`0xb1`	0	
Load Breakpoint 2 Response	`0xb2`	0	
Load Breakpoint 3 Response	`0xb3`	0	
Pause Execution Response	`0xf0`	0	
Resume Execution Response	`0xf2`	0	
Step Execution Response	`0xf3`	0	
Resume Execution Response	`0xf2`	0	
Move PC to Destination Response	`0xe3`	0	
Resume Unpipelined Response	`0xf5`	0	
Breakpoint Fired	`0xff`	8	64-bit Address

Table 6.2: Message types from the debug unit

Breakpoint Fired The “Breakpoint Fired” message is sent when an instruction commits a next PC in write-back which is equal to one of the four breakpoint registers. The “Breakpoint Fired” message has a payload containing the value of the breakpoint which fired which is an 8-byte address.

Chapter 7

The CHERI unit test suite

The CHERI prototype includes a simple unit test suite implemented using the Python Nose framework. The test suite exercises key CHERI functionality in a controlled and easily diagnosable environment, an instrumented CHERI simulator, with a goal of testing both basic MIPS ISA functionality and CHERI's security extensions. This chapter explains the structure and components of the test suite, how to run the test suite, how to add new tests, and some of the tools available for diagnosing test results.

7.1 The CHERI unit test environment

The CHERI unit test suite is implemented using a combination of the CHERI Bluespec simulator (with extensions for debugging), `make`, the MIPS toolchain, the Python Nose test framework, and a moderate collection of test programs and Nose classes to evaluate test output. The unit test suite can also be run against the `gxemul` MIPS simulator, which has proven useful for checking our interpretation of the MIPS ISA against a more common interpretation. In the future, we hope also to run the test suite against CHERI synthesised in an FPGA, likely with the help of JTAG.

7.1.1 CHERI test suite directory layout

Table 7.1.1 describes the directories in the CHERI unit test suite.

7.1.2 CHERI ISA extensions for testing

The CHERI test suite employs debugging extensions to the 64-bit MIPS ISA to examine the state of a simulated CHERI system after each test, dumping the general-purpose register file, the CP0 registers and the capability coprocessor registers and allowing tests to terminate the simulation in a controlled manner. Current extensions are exposed via CP0 register operations, although in the future it appears likely that they will move to capability coprocessor extensions to reduce the possibility of a collision with the existing MIPS ISA. In the future, we anticipate adding further extensions in support of testing to dump the simulation memory image.

Directory	Description
`cheritest/trunk/`	Root of the CHERI test suite tree, home of the makefile, linker scripts, and test library code
`gxemul_log`	Destination for gxemul test run output
`log/`	Destination for CHERI simulator test run output
`obj/`	Destination for test object files, memory images, and assembly dumps
`tests/`	Various subdirectories holding source code for individual tests, and their matching Python Nose classes
`tools/`	Utility functions to perform common functions such as interpreting CHERI simulator and `gxemul` output

Table 7.1: Directories in the CHERI unit test suite

7.1.3 Unit test support library

Most CHERI unit tests are linked against a thin "loader", `init.s`, which is responsible for setting up various aspects of CPU and memory configuration:

- Set up a stack at the top of memory.
- Install default before- and after-boot exception vectors and handlers, which will dump the register file and terminate if triggered.
- Explicitly clear all general-purpose registers except stack-related registers that may have been modified during startup.
- Invoke a user-provided `test` function using `JAL`; currently all test functions are implemented in assembly, but the calling convention should support C as well.
- On return from `test`, dumps the register file and terminates.

In addition, a small library of support routines common to more complex tests may be found in `lib.s`, including functions for copying memory and installing exception handlers. We anticipate that this library will grow in size as the test suite is made more comprehensive.

A few low-level tests, referred to as *raw tests*, execute directly rather than via `init.s`, and are not linked against `lib.s`. Raw tests perform low-level verification of CPU functionality required to reliably run `init.s`, such as initial register file values on CPU reset, arithmetic instructions, the reliability of branch and jump instructions, and basic memory operations. Whenever possible, it is best to avoid writing raw tests, which necessarily replicate functionality (such as register dumping), and lack access to a pre-configured stack.

Note that all tests will be run twice by the suite – once from uncached instruction memory, and once from cached instruction memory. Timing and pipeline effects differ significantly between the two cases. One impact of this is that all tests must be relocatable and able to run in multiple MIPS xkphys segments.

7.2 Running the CHERI test suite

Typically, the test suite will be run as follows:

```
$ cd ctsrd/cheritest/trunk
$ make test
```

The `CHERIROOT` variable may be used to tell the test suite where to find CHERI tools for processing memory images and the CHERI simulator; the CHERI simulator must first have been built using `make sim` or similar. The test suite may be run against `gxemul` as follows:

```
$ cd ctsrd/cheritest/trunk
$ make gxemul-build
$ make gxemul-nosetest
$ make gxemul-nosetest_cached
```

7.2.1 Jenkins

If you are developing in the Cambridge development environment, the CHERI unit test suite is run automatically by the Jenkins build framework. Jenkins can be monitored by visiting the following URL:

`https://ctsrd-build.cl.cam.ac.uk`

7.3 Unit test structure

Each unit test consists of a short assembly program exercising specific features in the CHERI CPU, and a Nose class that contains a set of assertions about termination state for the test. Modifications to the test suite typically take the form of modifying an existing test to check new assertions, or adding an entirely new test via a new test program and set of corresponding assertions.

7.3.1 Test types

Tests are split into two categories: raw tests that have few low-level dependencies and are intended to exercise basic CPU features such as the register file, and higher-level tests that are able to depend on common CPU initialisation code and a support library. Raw tests are run strictly before higher-level tests, which typically depend on features checked in raw tests. Raw test files are prefixed with `raw_`, and higher-level test file names are prefixed with `test_`; the build framework uses these prefixes to identify assembly and linking requirements, so they must be used.

Unless there is a specific reason to do so, new tests should be added as higher-level tests, relying on the `init.s` framework to set up the stack, dump register state on completion, and terminate the simulator, rather than hand-crafting this code. This provides access to routines such as `memcpy` that are frequently useful when implementing tests.

7.3.2 Test structure

All tests are compiled using 64-bit MIPS instructions, and attempt to follow a standard application binary interface (ABI) to allow existing compiled MIPS code to be easily reused in the test environment. Currently, no C code is linked into the test suite, but it is easy to imagine doing so in the future, making ABI conformance critical.

High-level tests implement a single, global function `test`. When `test` terminates, the calling code in `init.s` will dump register state and terminate the simulator; these registers then become available to the Nose test framework for checking. Other than changes to the program counter, **$PC**, the test framework avoids any changes to register values after the test returns. Tests may rely on the availability of a roughly 1K stack. Tests execute in both the cached but unmapped region of memory around `0x9800000040000000`, with a stack growing down from `0x9800000080008000`, and the uncached and unmapped region around `0x9000000040000000`, with a stack growing down from `0x9000000080000000`, but may make use of any required processor features such as cached and mapped memory regions, CP0 MMU operations, etc.

7.3.3 Test termination

Normally, high-level tests will terminate by returning from the `test` function, triggering a register dump and simulator termination. However, the test framework is executed with a 100000-cycle limit on simulation time in order to ensure termination, catching (for example) infinite loops in software, or exception cycles. As tests become more complicated, this limit may need to be changed, but is present to ensure that tests will eventually always terminate, even if software enters an infinite loop.

7.3.4 Connecting new tests to the build

Nose test files must begin with the prefix `test_`, which will normally occur for high-level tests; Nose test files for raw tests will therefore be prefixed with `test_raw_`. New unit tests are hooked up to the build system by adding their source files to the `TEST_FILES` variable in the makefile. This is normally done by adding the test filename to one of the make variables for a test subset such as `TEST_ALU_FILES`. For the time being, all test source and Nose files must be placed under the `tests` directory in an appropriate sub-directory which should be included in the `TESTDIRS` variable.

7.4 Example unit test: register zero

To explore the above design, we will consider the `test_reg_zero` unit test, which checks that the MIPS general-purpose register **R0**, also known as **$zero**, has the required special property that it always return the value 0. The correct functioning of **$zero** is not required for any raw tests, nor `init.s`, so the test is placed in the high-level test suite. The test performs a number of activities:

- Sets up a stack from for the function `test` by manipulating **$sp** and **$fp**.
- Pushes the return address, **$ra**, and saved frame pointer, **$fp**, onto the stack.
- Copies a value from **$zero** into **$t0** for inspection.
- Assigns a value to **$zero** from an immediate, and then copies out to **$t1** to confirm that the value does not get saved.
- Assigns a value to **$zero** from a register, and then copies out to **$t2** to confirm that the value does not get saved.
- Restores **$fp** and **$ra** from the stack and returns.

7.4.1 Register zero test code

Example assembly source code is illustrated in Figure 7.4.1.

7.4.2 Register zero Nose assertions

Figure 7.4.2 illustrates the Nose assertion set for this test, confirming a number of desired properties that should hold after the test code runs:

- that **$zero** held zero on exit
- that **$t0** held zero on exit, meaning that a simple move from **$zero** held zero on start
- and that registers **$t1** and **$t2** held zero values, meaning that various writes to **$zero** did not change the value returned when reading the register

7.5 Conclusion

This chapter has introduced the CHERI unit test suite, exploring both the structure of the suite and the implementation of individual tests. The test suite is intended to supplement formal methods by testing the programmer-level view of ISA correctness, and while it cannot be authoritative regarding the correctness of CHERI, it is extremely valuable in development, exercising critical instruction combinations and providing clear diagnostics. We hope to introduce a new unit test for each bug encountered in CHERI, and expand the test suite to provide detailed coverage of new ISA features.

```
.set mips64
.set noreorder
.set nobopt
.set noat

#
# This test checks that register zero behaves the wave it should: each of
# $t0, $t1, and $t2 should be zero as at the end, as well as $zero.
#

.global test
test: .ent test
daddu  $sp, $sp, -32
sd $ra, 24($sp)
sd $fp, 16($sp)
daddu $fp, $sp, 32

# Pull an initial value out
move $t0, $zero

# Try storing a value into it from an immediate
li $zero, 1
move $t1, $zero

# Try storing a value into it from a temporary register
li $t3, 1
move $zero, $t3
move $t2, $zero

ld $fp, 16($sp)
ld $ra, 24($sp)
daddu $sp, $sp, 32
jr $ra
nop # branch-delay slot
.end test
```

Figure 7.1: Example regression test checking properties of **$zero**

```
from bsim_utils import BaseBsimTestCase

class test_reg_zero(BaseBsimTestCase):
    def test_zero(self):
        '''Test that register zero is zero'''
        self.assertRegisterEqual(self.MIPS.zero, 0)

    def test_t0(self):
        '''Test that move from zero is zero'''
        self.assertRegisterEqual(self.MIPS.t0, 0)

    def test_t1(self):
        '''Test that immediate store of non-zero to zero returns zero'''
        self.assertRegisterEqual(self.MIPS.t1, 0)

    def test_t2(self):
        '''Test that register store of nonzero to zero returns zero'''
        self.assertRegisterEqual(self.MIPS.t2, 0)
```

Figure 7.2: Example Nose assertion file for the **$zero** test

Chapter 8

CHERI on Altera FPGAs

WARNING: the contents of this chapter are frequently out of date as a result of evolving Altera tools and changes to the CHERI build environment.

This chapter describes how to build CHERI for synthesis using Bluespec, configure the Altera build environment, and synthesise CHERI for the Terasic tPad and DE4 FPGA teaching boards described in later chapters. This information is of relevance to researchers working with the CHERI hardware design; software consumers of CHERI can find information on using specific Terasic boards in Chapter 9, and will not need to follow the directions in this chapter.

8.1 Building CHERI for synthesis

It is possible to compile the CHERI source code to Verilog with two targets as shown in Table 8.1

The CHERI verilog build is also sensitive to two make variables described in Section 5.3. The result of building CHERI for synthesis is a set of Verilog files in `verilog/generated/`, with the file `mkTopAvalonPhy.v` containing the top-level module. These files may be copied into one of the directories in the `boards/` directory to be synthesized for a particular board. As a note, when a supported FPGA board is connected to the computer by USB, it is possible to connect to the CHERI UART using JTAG:

```
$ nios2-terminal --instance 0
```

Target	CP2
`verilog`	No
`verilog_cap`	Yes

Table 8.1: CHERI verilog build targets

8.2 The Altera development environment

Terasic's FPGA evaluation boards include Altera FPGAs; the following sections depend on the correct installation of Altera's FPGA development toolchain in order to synthesise and program the on-board FPGAs. Some of Altera's tools, especially the GUIs but also some command-line tools, require X11; in these cases, if using a central build server, ensure that the -X argument is passed to the `ssh` command:

```
$ ssh -X user@zenith.cl.cam.ac.uk
```

To configure your shell to use Bluespec, Altera, and other development toolchain elements for CHERI, such as compilers and linkers, use the following script from the CTSRD Subversion repository (described in previous chapters):

```
$ cd ctsrd/cheri/trunk
$ source setup.sh
```

In order to successfully build the CHERI hardware project, ensure that you have added the Bluespec Verilog library to the Quartus global library path. This is typically located at:

```
<BluespecDirectory>/lib/Verilog
```

Also ensure that you have added any relevant license files needed to build the project. For example, if you are using an Terasic touchscreen, you may need to add the license file for the `i2c_touch` Verilog module to the license file string for Quartus.

The Quartus 12.1 tools assume that /bin/sh is a link to /bin/bash but on recent Ubuntu systems the symbolic link is to /bin/dash. This breaks some of the Qsys generate scripts. There are two solutions:

1. Patch the Quartus 12.1 distribution so that scripts starting:

   ```
   #!/bin/sh
   ```

   now start

   ```
   #!/bin/bash
   ```

   This is what has been done in the Computer Lab.

2. Modify the symbolic link so that /bin/sh links to /bin/bash. Although we've never found a problem with this solutions, it seems inelegant and runs the risk of breaking some aspect of your Linux setup.

Finally, if you are using Ubuntu, you may need to insert a new rules file into `/etc/udev/rules.d/`. You might add a new file named `51-usbblaster.rules` with the following contents:

Directory	Board
`ctsrd/cheri/trunk/boards/terasic_tPad`	Terasic tPad
`ctsrd/cheri/trunk/boards/terasic_de4`	Terasic DE-4

Table 8.2: Terasic per-board directories

Target	Description
`all`	builds everything using the following steps (except download)
`build_cheri`	build the CHERI processor
`build_peripherals`	build the peripherals
`build_miniboot`	build miniboot ROM and copy initial.hex here
`build_qsys`	build Qsys project containing CHERI, etc.
`build_fpga`	synthesis, map, fit, timing analyse and generate FPGA image
`report_critical`	scans build_fpga reports for critical warnings
`report_error`	scans build_fpga reports for errors
`download`	attempts to download the FPGA (.sof) image to the FPGA but the chain file (.cdf) may need to be updated for your configuration (e.g. USB port number)
`clean`	remove Quartus and Qsys build files
`cleanall`	clean + clean peripherals, CHERI and miniboot

Table 8.3: Make targets for per-board directories

```
# Set permissions for Altera USB Blaster
SUBSYSTEM=="usb", ATTR{idVendor}=="09fb", ATTR{idProduct}=="6001", \
MODE="0666", OWNER="root", GROUP="dialout"
# Set permissions for Fast Altera USB2 Blaster
SUBSYSTEM=="usb", ATTR{idVendor}=="09fb", ATTR{idProduct}=="6810", \
MODE="0666", OWNER="root", GROUP="dialout"
```

8.3 Synthesising CHERI

WARNING: this applies to the DE-4 board setup but the tPad setup has atrophied and needs attention.

The CTSRD project provides reference configurations for CHERI on the Terasic tPad and DE-4 boards; per-board directories are listed in Table 8.2. Each board directory contains its own Quartus project, `Makefile`, etc. Table 8.3 shows the available `make` targets.

Targets `build_cheri` and `build_peripherals` cause other Makefiles to be used to build various Verilog components that are found by Quartus via the paths in `peripherals.ipx` and `processors.ipx`. `build_miniboot` compiles the miniboot loader C code and produces a ROM image initial.hex which is copied into the board directory.

A `make cleanall;make all` will take around 40 minutes to an hour to complete. To then download the system to an FPGA board plugged into the host PC, `make download` may work though a new chain (.cdf) file might be needed to reflect your particular setup. This is most easily achieved using the GUI in Quartus to open the programmer and configure.

Chapter 9

CHERI on Terasic tPad and DE4

This chapter describes how to use the CHERI processor prototype on the Terasic tPad and DE4 FPGA teaching boards. This includes both tutorial material on programming the board, and material on how board peripherals are exposed to CHERI in the reference designs provided by the CTSRD project. The chapter is intended to support software development on CHERI; see Chapter 8 for documentation pertinent to hardware development. Later chapters describe building and using CheriBSD on Terasic boards.

9.1 CHERI configuration on the tPad and DE4

Communication with external I/O devices, such as NICs, is accomplished via a blend of memory-mapped I/O, interrupts, and (eventually) DMA. The CHERI processor and operating system stack supports a variety of peripherals ranging from Altera "soft" cores, such as the JTAG UART and SD Card IP cores, to "hard" peripherals provided by Terasic on its tPad and DE4 development boards. The following sections document available peripherals and their configuration on the Avalon system-on-chip bus as configured in the CHERI reference designs.

9.1.1 Physical address space on the tPad

Table 9.1 shows the physical addresses reserved for I/O devices in the CHERI reference tPad configuration.

9.1.2 Physical address space on the DE4

Table 9.2 shows the physical addresses reserved for I/O devices in the CHERI reference DE4 configuration.

Base address	Length	IRQ	Description
`0x7f000000`	64	0	Altera JTAG UART
`0x7f001000`	64	-	Altera JTAG UART for debugging output
`0x7f002000`	64	-	Altera JTAG UART for data
`0x7f004000`	4	-	Old location of count register until 2013-03-01
`0x7f007000`	1024	-	Altera Triple-Speed Ethernet MegaCore MAC control
`0x7f007400`	8	-	... MAC transmit FIFO
`0x7f007420`	32	2	... MAC transmit FIFO control[1]
`0x7f007500`	8	-	... MAC receive FIFO
`0x7f007520`	32	1	... MAC receive FIFO control[2]
`0x7f008000`	1024	3	Altera University Program Secure Data Card IP Core
`0x7f00A000`	20	-	Hardware Version ROM [3]
`0x7f800000`	8 MB	-	Bluespec Peripheral Address Space
`0x7f800000`	8	-	... Count Register (from 2013-03-01)
`0x7f804000`	16 KB	-	... CHERI PIC_CONFIG_BASE

[1] See "Avalon-MM Write Slave to Avalon-ST Source"
[2] See "Avalon-ST Sink to Avalon-MM Read Slave"
[3] See Table 9.3

Table 9.1: Bus configuration for CHERI's reference tPad configuration

Base address	Length	IRQ	Description
`0x70000000`	128 MB	-	Cambridge Multitouch LCD + 256 Mb Intel StrataFlash
`0x7f000000`	64	0	Altera JTAG UART
`0x7f001000`	64	-	Altera JTAG UART for debugging output
`0x7f002000`	64	-	Altera JTAG UART for data
`0x7f002100`	32	-	Altera UART (RS232) [0]
`0x7f004000`	4	-	Old location of count register until 2013-03-01
`0x7f005000`	1024	-	Altera Triple-Speed Ethernet MegaCore MAC control Port 1
`0x7f005400`	8	-	... MAC transmit FIFO
`0x7f005420`	32	2	... MAC transmit FIFO control[1]
`0x7f005500`	8	-	... MAC receive FIFO
`0x7f005520`	32	1	... MAC receive FIFO control[2]
`0x7f006000`	1	-	DE4 LEDs, one bit per LED
`0x7f007000`	1024	-	Altera Triple-Speed Ethernet MegaCore MAC control Port 0
`0x7f007400`	8	-	... MAC transmit FIFO
`0x7f007420`	32	2	... MAC transmit FIFO control[1]
`0x7f007500`	8	-	... MAC receive FIFO
`0x7f007520`	32	1	... MAC receive FIFO control[2]
`0x7f008000`	1024	3	Altera University Program Secure Data Card IP Core
`0x7f009000`	2	-	Switches and Buttons one bit each (DIP[0:7], SW[0:3], BUTTON[0:3])
`0x7f00A000`	20	-	Hardware Version ROM [3]
`0x7f00B000`	8	-	OpenCores i2c Controller for the HDMI chip
`0x7f00B080`	1	-	1-bit PIO to reset the HDMI chip
`0x7f00C000`	8	-	Temperature and Fan Control [4]
`0x7f100000`	256 KB	-	Philips ISP1761 USB 2.0 Chip [5]
`0x7f800000`	8 MB	-	Bluespec Peripheral Address Space
`0x7f800000`	8	-	... Count Register (from 2013-03-01)
`0x7f804000`	16 KB	-	... CHERI PIC_CONFIG_BASE

[0] See “UART Core” in the Altera “Embedded Peripherals IP User Guide”
[1] See “Avalon-MM Write Slave to Avalon-ST Source”
[2] See “Avalon-ST Sink to Avalon-MM Read Slave”
[3] See Table 9.3
[4] See Section 9.5
[5] See Philips ISP1761 Hi-Speed Universal Serial Bus On-The-Go controller datasheet, available online

Table 9.2: Bus configuration for CHERI’s reference DE4 configuration

Base	Length	Item	Format
`0x0`	4 Bytes	Build Date	Binary coded decimal formated as mmddyyyy
`0x4`	4 Bytes	Build Time	Binary coded decimal formated as 00hhmmss
`0x8`	4 Bytes	Svn Version	Binary coded decimal
`0xc`	8 Bytes	Host Name	ASCII, truncated to 8 characters

Table 9.3: Contents of the CHERI Hardware Build Version Number ROM

9.2 Altera IP cores

CHERI and FreeBSD support a number of Altera "soft" IP cores on the Terasic tPad and DE4 platforms. Many of these IP cores are documented in the *Embedded Peripherals IP User Guide*[1] provided by Altera, including the JTAG UART core and Avalon-MM and Avalon-ST bus attachments.

Certain Altera IP cores are described in other documents, including the Altera Triple-Speed MAC described in the *Triple-Speed Ethernet MegaCore Function User Guide*[2], and SD Card IP core described in the *Altera University Program Secure Data Card IP Core*[3] documents from Altera.

9.3 Cambridge IP cores

Cambridge provides two "soft" peripheral devices: a *count device*, which simply provides a memory-mapped register that is incremented on every read, intended for cache testing, and a memory-mapped interface to the Terasic MTL multitouch LCD panel. This latter IP core includes both memory-mapped support for a pixel frame, and a VGA-like text frame buffer suitable for use as a system console. It also provides access to multitouch input.

9.3.1 The DE4 multitouch LCD

Hardware overview

A Terasic MTL-LCD is connected to the DE4 via the supplied ribbon cable. This provides a parallel interface running at 33 MHz to drive the LCD and an I2C interface to obtain touch information.

[1]`http://www.altera.com/literature/ug/ug_embedded_ip.pdf`

[2]`http://www.altera.com/literature/ug/ug_ethernet.pdf`

[3]`ftp://ftp.altera.com/up/pub/Altera_Material/11.0/University_Program_IP_Cores/Memory/SD_Card_Interface_for_SoPC_Builder.pdf`

Terasic provides an encrypted block (`i2c_touch_config.v`) to talk I2C to the touch panel and exports parameters as a simple parallel interface.

We have built three key hardware components to interface to the MTL-LCD:

MTL_LCD_Driver – This peripheral takes an AvalonStream of pixel values and maps them to the MTL (multi-touch) LCD colour screen, which has an 800x480 resolution. Pixels are 24-bits (8-bit red, green, blue). The main clock must run at the pixel clock rate of 33 MHz. The clock to the MTL-LCD (`mtl_dclk`) must be fed to the LCD outside of this module. A dual-clock FIFO is needed in the AvalonStream between this module and the MLT_Framebuffer_Flash.

MTL_LCD_HDMI – This peripheral is an alternative to MTL_LCD_Driver which runs the multitouch LCD out of spec. (but still working just fine) in order to mirror to HDMI (and via HDMI to VGA) at 720x480 pixels with the correct timing specification. H-sync and V-sync timings are changed and the pixel clock is reduced to 27MHz. This reduced pixel clock rate has the advantage that the bandwidth from the SSRAM frame buffer memory is less demanding. As with the MTL_LCD_Driver, this module is connected via a dual-clock FIFO and an AvalonStream interface to the MTL_Framebuffer_Flash. No changes are needed to MTL_Framebuffer_Flash to use this module since the difference in pixel clock rate is accommodated by flow-control in the AvalonStream.

MTL_Framebuffer_Flash – Provides a memory-mapped frame buffer using the DE4's off-chip SSRAM to store the frame buffer and provides access to the Flash, which is on the same bus as the SSRAM. Provides an Avalon memory mapped interface to allow a processor to write to the SSRAM. This module is designed to work at the main system clock rate of 100 MHz. Note that the clock to the SSRAM needs to be provided outside of this module, direct from a PLL. The SSRAM conduit interface must be connected to the SSRAM pins. The I2C conduit interface (coe_touch) must be connected to Terasic's I2C encrypted block outside of the Qsys project.

In addition, the following libraries of ours are used:

AlteraROM – provides a font ROM initialised from fontrom.mif

VerilogAlteraROM.v – Verilog wrapped by AlteraROM

Avalon2ClientServer – provides the Avalon memory-mapped interface

AvalonStreaming – provides the Avalon streaming interface

Software overview

The MTL_Framebuffer is accessed via an 8 MB memory mapped region where the first 2 MB maps the SSRAM, which contains both the text and pixel frame buffer. Control registers start 4 MB into the region. Random access reads and writes of arbitrary size are permitted to the main frame buffer, but registers require 32-bit accesses. Note that writes to the frame buffer are queued and incur little latency whereas reads need to schedule access around the LCD updates so incur a much greater latency penalty. Reads and writes to registers are quick.

The pixel frame buffer is 32 bits per pixel with the upper byte being ignored followed by bytes of red, green and blue channels. The resolution is 800x480 with the first pixel being top level. The text frame buffer accepts characters of 16-bits with the upper byte representing the VGA text colour and the lower byte holding the character. There are 100 columns and 40 rows of text. VGA text colour is a byte in the following format: (1-bit flashing, 3-bit background colour, 4-bit foreground colour). Colours are from the following table:

code	**colour**	**code**	**colour**
0	black	8	dark grey
1	dark blue	9	light blue
2	dark green	10	light green
3	dark cyan	11	light cyan
4	dark red	12	light red
5	dark magenta	13	light magenta
6	brown	14	light yellow
7	light grey	15	white

See mtl_test_small.c for an example which drives the MTL-LCD using a NIOS for some helper functions, and so on. Table 9.4 describes the memory map of the MTL-LCD.

The frame-buffer-blending register has the following format (from MSB to LSB):

- Top 4 bits are unused but should be set to zero.
- 4-bits of VGA colour code providing a default colour for the whole screen. After reset, this is set to 2, i.e., dark green. Typically, this will need to be set to 0 for general use.
- 8 bits of alpha blending for the pixel frame buffer. This value is subtracted using saturation arithmetic from the character colours, so a value of 255 erases the character frame buffer. Reset value is 255, i.e., characters off.
- 8 bits of alpha blending for the character frame buffer foreground colour, subtracted from the pixel background colour using saturation arithmetic. So 255 makes the characters opaque and 0 makes them transparent. Reset value is 255, i.e., pixel off when the character pixel is on.
- 8 bits of alpha blending for the character frame buffer background colour which is subtracted from the pixel colour using saturation arithmetic. So 255 makes the background opaque and 0 makes the background transparent. Set to 255 after reset, i.e., pixel off when the character pixel is off.

The MTL two-touch gesture codes (copied from the MTL-LCD manual):

code	**gesture**
0x30	north
0x32	north-east
0x34	east
0x36	south-east
0x38	south
0x3A	south-west
0x3C	west
0x3E	north-west
0x40	click
0x48	zoom in
0x49	zoom out

base offset	**length**	**description**
0x0000000	2MB	SSRAM
0x0000000	800x480 words	pixel frame buffer, 32-bit colour, although only 24 bits used: 8 bits each of (r,g,b) where b is the LSB
0x0177000	100x40x2 bytes	of text buffer in its default location
0x0400000	1 word	frame buffer blending (see below)
0x0400004	1 word	text cursor position, bytes: (unused, unused, x, y)
0x0400008	1 word	character frame buffer offset base address relative to the start of the SSRAM, 0x177000 after reset (i.e., 800x480 words, so just after the pixel buffer). Note that this must be a 32-bit word aligned offset.
0x040000c	1 word	touch point x1, -1 if not valid
0x0400010	1 word	touch point y1, -1 if not valid
0x0400014	1 word	touch point x2, -1 if not valid
0x0400018	1 word	touch point y2, -1 if not valid
0x040001c	1 word	(touch_count,gesture), -1 if not valid, where touch_count is a 2-bit value (0,1 or 2 touches) and gesture is an 8-bit value (see table below for details). Reading this register dequeues all of the current touch sensor values.
0x4000000	64MB	Flash memory (see below)

Table 9.4: Memory map used for the MTL device

9.3.2 HDMI chip configuration via I2C

We use the Terasic HDMI_TX_HSMC daughter card on the DE4 board to obtain HDMI output mirroring. Pixel data, H-sync and V-sync are provided by MTL_LCD_HDMI (see Section 9.3.1) when mirroring the multitouch LCD. However, to obtain output it is necessary to configure the HDMI chip on the daughter card via an I2C interface. To do this we use the I2C master interface from OpenCores[4]. This is wrapped in an Avalon interface we have written[5] which is colocated with documentation[6]

Currently miniboot uses this I2C interface to configure the HDMI chip to a fixed output resolution. The code was hacked up in time for the November 2012 PI meeting and will need further work when we want to dynamically change the display resolution, etc. But first we'll need a frame buffer which can produce programmable resolutions.

9.4 Standalone HDMI output

The standalone HDMI output (HDMI_Driver) is an alternative to the mirrored HDMI output from the MTL-LCD discussed in Section 9.3.2. The motivation is to provide support for video streams of different resolutions from other sources (e.g. streaming out of high-bandwidth memory like DDR2 memory).

In order to support multiple resolutions, a variable pixel clock is required (Section 9.4.1) together with a software configurable HDMI timing generator (Section 9.4.2) and the HDMI chip configuration via I2C discussed earlier in Section 9.3.2. Note that we currently using the I2C interface to place the HDMI chip into DVI compatibility mode. In this mode it appears that the resolution can be set by changing the pixel clock frequency and video timing (sync signals) without further configuration of the HDMI chip. The HDMI chip documentation is so poor that it's difficult to determine whether this is the correct usage, but it appears to work.

9.4.1 Reconfigurable Video Pixel Clock

This is a simple Qsys peripheral written in SystemVerilog to provide an Avalon memory mapped interface to Altera provided reconfigurable PLL. The reconfigurable PLL needs to be instantiated outside of this module using an ALTPLL megafunction with its reconfiguration interface enabled.

Inside this peripheral an ALTPLL_RECONFIG is instantiated which provides a cache of the PLL parameters and when triggered it the writes them to the ALTPLL using a proprietary serial interface. ALTPLL_RECONFIG also resets the ALTPLL post configuration.

This module is addressed as follows. **All addresses refer to 32-bit little endian words. Byte addressing is not supported.**

The lower address bits have the following meaning:

[4] `http://opencores.org/project,i2c`

[5] `cherilibs/trunk/peripherals/i2c/i2c_avalon.sv`

[6] `cherilibs/trunk/peripherals/i2c/i2c_rev03.pdf`

bits	1-0	are always zero (word aligned)
bits	5-2	is the counter_type
bits	8-6	is the counter_parameter
bit	9	When=0 it refers to the ALTPLL_RECONFIG parameters (above). When =1 for a write it causes the PLL parameters to be written to the PLL. When =1 and reading, it returns busy=-1, done=0.

counter_type and counter_parameters are defined in Altera's ALTPLL_RECONFIG Users Guide[7] with the parameters for Stratix IV PLLs appearing on pages 45–46.

For Stratix IV parts (e.g. on the DE4 board) the following counter_parameters are particularly useful:

counter_parameter number	variable name	meaning
0	n	master multiplier
1	m	master divisor
4	c0	further divisor for clock 0

The output frequency clock c0 is given by: fout_c0 = (n $\times$ fin) / (m $\times$ c0)

Where fout_c0 is the output frequency for clock 0 on the PLL and fin is the input clock frequency (typically from an external pin on the DE4 board running at 50MHz).

For each of these counter_parameters, the following counter_types need to be set (e.g. for a required value v where v>0):

counter_type number	variable name	bit width	value from v
0	high_count	(9-bits)	(v+1)/2
1	low_count	(9-bits)	v – high_count
4	bypass	(1-bit)	(v==1) ? 1 : 0
5	odd_count	(1-bit)	v & 0x1

After setting the above parameters in the ALTPLL_RECONFIG cache you need to trigger it to write them to the ALTPLL by writing some word of data (the data is irrelevant) to an address on this peripheral with address bit 9 set.

9.4.2 HDMI Timing Driver

This Qsys peripheral (HDMI_Driver) takes an AvalonStream of pixel values and maps them to the Terasic HDMI Transmitter daughter card (HDMI_TX_HSMC). It needs to be clocked at the video pixel clock frequency which may be variable so an Avalon clock crossing bridge is needed to interface to the AvalonMM slave interface which allows the following parameters to be set from software.

[7] `http://www.altera.co.uk/literature/ug/ug_altpll_reconfig.pdf`

Address map (32-bit word offset, little endian 12-bit values in 32-bit word)		
0	x-resolution	(in pixels)
1	horizontal pulse width	(in pixel clock ticks)
2	horizontal back porch	(in pixel clock ticks)
3	horizontal front porch	(in pixel clock ticks)
4	y-resolution	(in pixels/lines)
5	vertical pulse width	(in lines)
6	vertical back porch	(in lines)
7	vertical front porch	(in lines)

9.5 Temperature and fan control

The temperature and fan control peripheral has two read-only 32-bit registers. The first (address 0x0) returns the last temperature reading as a 32-bit signed integer in degrees Centigrade. The second (address 0x4) is the power to the fan as a range from 0 to 255.

9.6 Terasic hard peripherals

9.6.1 Intel StrataFlash 64M NOR flash

The DE4 board has a single Intel StrataFlash embedded memory and we have the part with 64 MB (512 Mb), which is 16 bits wide. Note that this part might be in one packaged, but it has two die stacked flash chips internally and they work independently. This flash memory sits on the same bus as the SSRAM used for the frame buffer so the memory transactions are handled by the multitouch display hardware.

Read mode

After reset, the memory is in read mode, and memory read accesses (bytes, 16-bit and 32-bit word) accesses appear like conventional memory. After being in another mode, read mode can be enabled by writing 0x00ff (little endian) or 0xff00 (big endian) to the base address.

Write mode

But writes are treated as commands, not memory writes. This is where it gets a lot more complicated and reading the data sheet is required. But here are some notes.

Two chips – The DE4 has a 512 Mb part containing two 256 Mb dies (chips) in the same package. Therefore there are actually two independent devices so, for example, reading status is on a per-die basis. Address bit 25 determines which die is being talked to.

Data width – The device is 16 bits wide, and byte-wide accesses make no sense to it. Only use 16-bit writes. 32-bit reads will be turned into two 16-bit reads by our hardware.

Block sizes – Each flash chip is broken down into programming regions and blocks. Blocks are not equal in size. Blocks 0 to 254 are 128 KB in size and blocks 255 to 258 are 32 KB. See Table 7 on page 24 of the data sheet for further details.

Block erase – Data can be erased (set to 0xffff) only by erasing a whole block.

Write protect – After reset the flash part write protects the blocks. Software can issue a block unlock request before doing a write and then lock the block again afterwards. There are also one-time programmable lock registers and we suggest that you avoid touching these!

Writing data – Once a block is unlocked, data can be written one 16-bit word at a time by issuing a write command followed by the data. After doing a write, the status must be polled to determine when the write is complete before attempting another write or read. Writes can only clear bits; therefore, an erase may be necessary to set all of the bits in the block before doing the write.

Buffered writes – Writes can be conducted in blocks as large as 32 $\times$ 16-bit words. This is faster than single writes.

Here are some example access commands (see Table 21, page 51 of the data sheet for further details). Note that this assumes a **little endian view**:

Read mode (i.e., the same mode as after reset)

read/write	address	data	comment
write	base address	0x00ff	clear the status register

Unlock block for writes

read/write	address	data	comment
write	address within block	0x0060	unlock setup
write	address within block	0x00d0	unlock block

Lock block to write protect

read/write	address	data	comment
write	address within block	0x0060	unlock setup
write	address within block	0x0001	lock block

Status register

Notes on the status register based on Table 28, page 75 of the data sheet:

bit	name	meaning
7	device write status	0=busy, 1=ready
6	erase suspend status	erase suspend 1=not in effect, 0=in effect
5	erase status	0=success, 1=fail
4	program status	0=success, 1=fail
3	V_{pp} status	programming voltage status (0=good, 1=bad)
2	program suspend status	program suspend 1=not in effect, 0=in effect
1	block-locked status	block (0=not) locked during program or erase
0	BEFP status	see data sheet

Bits 7,6,2 are set and cleared by the flash write state machine but bits 5, 4, 3, 1 are only set by it, so a clear is needed before using them to check error status.

Note that these tables assume a **little endian view**:

Clear status register

read/write	address	data	comment
write	base address	0x0050	clear the status register

Read the status register

read/write	address	data	comment
write	base address	0x0070	read status register mode
read	base address	-	status register returned

Erase block

Erasing the block requires the following sequence (in pseudo code):

```
unlock_block_for_writes(offset)
clear_status_register
issue_erase_block_command(offset)
while (read_status_register = busy) {} // several million polls
read_status_register to see if erase passed
lock_block_to_prevent_writes
read_mode
```

Note that this table assumes a **little endian view**:

Erase unlocked block

read/write	address	data	comment
write	address within block	0x0020	block erase setup
write	address within block	0x00d0	block erase confirm

To erase a region of memory, the easiest thing seems to be to scan the memory to see if it contains 0xffff and when it does not, issue a block erase command at that address.

Note that Intel certify the part for a minimum of **100,000 erase cycles per block**.

Writes

Write sequence (post erase) starts with an unlock of the block, performing each write followed by a status register check, and finally locking the block again, and putting the device back into read mode (as above). The write component is performed using the following sequence (note that this table assumes a **little endian view**):

Write data

read/write	address	data	comment
write	address	0x0040	write command
write	address	data	write the data

Then pole the status register (see below) until bit 7 has gone to 1=ready. This typically took 52 polling loop iterations on a NIOS processor running at 100 MHz with each flash access taking 16 clock cycles, i.e., this is not fast!

Buffered Writes

Buffered writes are more efficient than single writes. The write sequence is only slightly more involved.

Buffered write data

read/write	address	data	comment
write	address	0x00e8	buffered write command
read	address	status	sr[7]=0 indicates failure
write	address	0x001f	number of data items to write minus one
write	address	data	write 32 words of data
write	address	0x00d0	confirm write
read	address	status	sr[7]=0 means busy, wait

Flash Regions

Terasic specifies uses for most of the flash memory in the *Terasic DE4 Getting Started Guide*. Some of these regions must remain used for their reserved purpose while others have been reallocated for other uses.

In our design, three regions are of particular importance. The region `0x00000000-0x00020000` is reserved for Terasic uses. The region `0x00020000-0x0181FFFF`is dedicated to two FPGA images, the first of which is loaded at power-up. This offset is programmed into the MAXII FPGA on the DE4 and cannot be changed easily. The region `0x02000000-0x03FFFFFF` (the entire second flash chip) is dedicated to a default software image to be relocated to DRAM at bootup. This is performed by the miniboot bootloader that is embedded in the FPGA image.

Table 9.5 lists our uses for each range and the corresponding FreeBSD device providing access the region. Changes to these allocations may require changes to this document, miniboot, cherictl,

offset range	CHERI use	device
0x00000000 - 0x00007FFF	user design reset vector	/dev/cfid0s.config
0x00008000 - 0x0000FFFF	ethernet option bits	/dev/cfid0s.config
0x00010000 - 0x00017FFF	board information	/dev/cfid0s.config
0x00018000 - 0x0001FFFF	PFL option bits	/dev/cfid0s.config
0x00020000 - 0x00c1FFFF	FPGA image 1 (power up)	/dev/cfid0s.fpga0
0x00c20000 - 0x0181FFFF	FPGA image 2 (on RE_CONFIGn button)	/dev/cfid1s.fpga1
0x01820000 - 0x03FDFFFF	operating system area	/dev/cfid0s.os
0x02000000 - 0x03FDFFFF	kernel (temporary)	/dev/map/kernel
0x03FE0000 - 0x03FFFFFF	boot loader	/dev/cfid0s.boot

Table 9.5: Layout of the on-board DE4 Intel StrataFlash

BERI_DE4.hints, and flashit.sh.

If portions of the flash are accidentally erased causing unexpected behaviour, factory behaviour can be restored by extracting and writing the file `cfi0-de4-terasic.bz` /dev/cfid0. This file can be found under `/usr/groups/ctsrd/cheri` on Cambridge systems.

```
# bunzip -c cfi0-de4-terasic.bz2 | dd of=/dev/cfid0
```

Hardware notes

The device comes out of reset in asynchronous mode operation, which seems to be easiest to deal with. So the clock to the flash device is simply kept at 0.

The bus is simple to use. Address, address-valid, chip-enable, write-enable, and output-enable can be asserted together. Writes take a minimum of 85 ns and the address and data are latched on the rising edge of write-enable. It seems to be a good idea to deassert chip-enable (i.e., set to 1) between each access to guarantee updates.