A MIPS R3000 microprocessor on an FPGA

Charles Brej

1 Introduction

The aim of the project was to implement a complete processor that is still widely used today. Of the many potential processors that were considered the one that was chosen was a MIPS R3000 microprocessor due to its simple instruction encodings. The R3000 microprocessor is not just a processor as it also includes cache, memory management and a coprocessor interface. (It is also capable of handling exact exceptions)

1.1 History of RISC

When the first processors and Instruction Set Architectures (ISAs) were created programming was very difficult and so complex instructions that looked more like high level languages were added to each ISA family. To allow old software to run on newer computers the instruction sets were expanded and became more and more complex. To add to the complexity issue memory prices were very high and to fit a program into a very small space it was necessary to create instructions with varying lengths. Also memory was a lot faster than the processor and so keeping all variables in memory was quite logical. In a completely different direction to the CISC movement of increasingly complex instruction sets the RISC adopted a totally different approach. By keeping all instructions the same size decoding becomes simple. Having a large register bank decreases memory accesses which are increasingly slow compared to the CPU speed on modern computers. Implementing only simple and common operations the processor speed can be increased and chip area can be decreased. Having a much simpler base architecture allows the CPU to implement other speed increasing methods much more simply for example pipelining.

1.2 Load Store Architecture

MIPS microprocessor instructions only allow registers or small immediates to be operands of operations. CISC on the other hand often has instructions which use memory stored data as operands. This makes the execution harder. For example the 8086 instruction 'CMP AX, ES:[SI+02]' firstly requires 2 to be added to SI and the result to be added to ES shifted by 4 this creating an effective address to load a 16 bit word (possibly non-word aligned so multiple loads are required) from memory then compare it to AX and write the flags created by the comparison to the flags register for use by the next conditional jump instruction. This scheme is difficult to implement and the instruction goes several times through the ALU. MIPS microprocessor instructions only go through the ALU once and never after a memory access. If an operand from memory is required it is loaded into the register bank first and only then used in subsequent operations. This allows the creation of a very simple architecture which is easy to

pipeline. A typical MIPS microprocessor uses a five stage execution pipeline as shown in Figure 1 on page 2.

FIGURE 1. Five stage pipeline

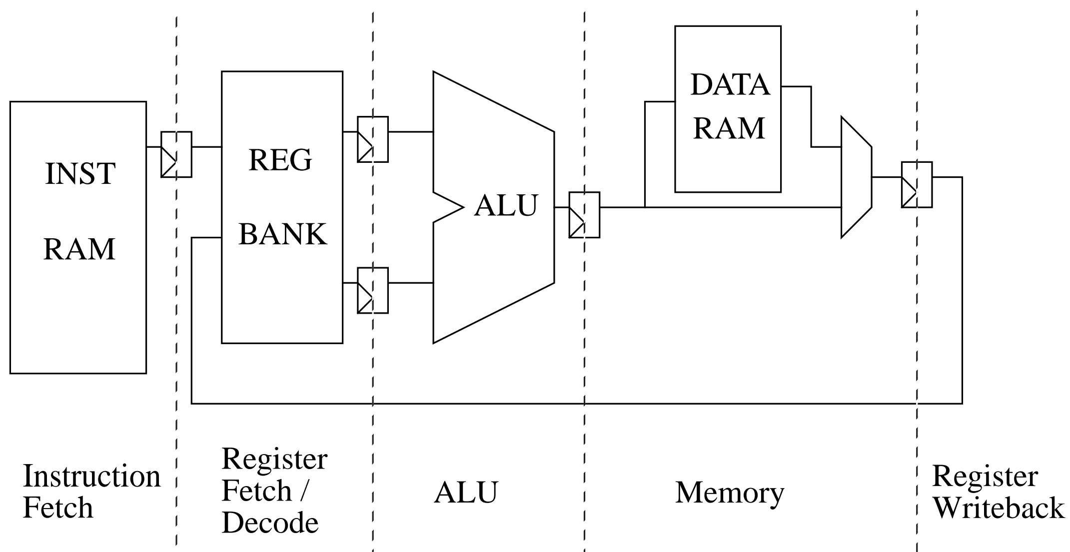
Instruction Fetch	Register Fetch / Decode	ALU	Memory	Register Writeback

In a five stage pipeline there are two memory ports: one for instruction fetch and one for data accesses. To deal with this either two separate memories can be used or swap memory access between the two ports. This is called Harvord architecture.

1.3 History of pipelining

Pipelining is a method of getting more than one instruction to execute simultaneously. By dividing the path that the instruction has to go through in the CPU into segments and placing latches at the beginning of each segment instructions will take several clocks to execute instead of one. But as MIPS microprocessor instructions only visit each segment once they only occupy one segment allowing other instructions to come straight after them and occupy other segments.

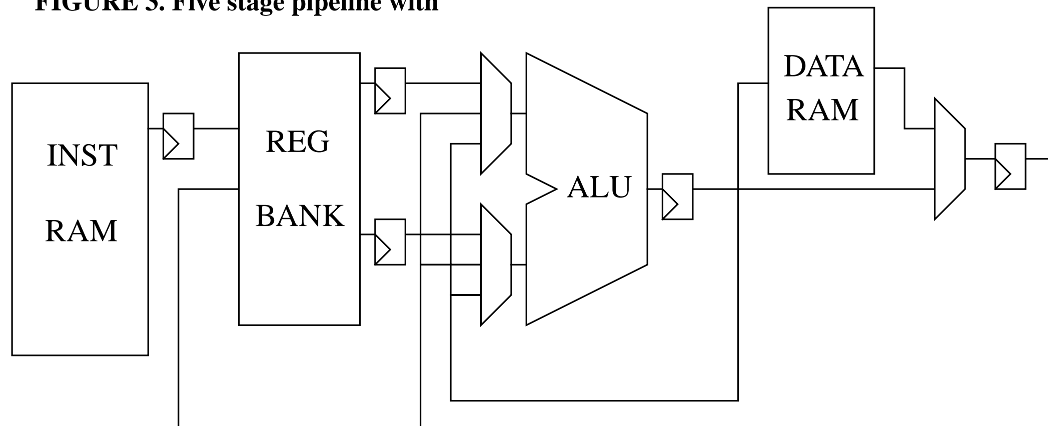
FIGURE 2. Five stage pipeline datapath



There are problems that arise with pipelining. Firstly if an ALU instruction writes to a register that is required in the next instruction the data in the register bank is not yet updated when the second instruction requests it as the data is now at the end of the ALU stage. The easiest way of getting the data back to the next instruction is to pick off the result from the ALU stage and replace the register bank value with it. The same can be done for data that is at the end of the memory stage. This still does not solve the problem of using a result from a memory operation on the next cycle. This can be

solved by the processor inserting a NOP instruction if it detects a dependency or the compiler simply never using a result from a memory operation on the next cycle.

FIGURE 3. Five stage pipeline with



1.4 Powerview

To design a MIPS microprocessor compatable CPU a schematic entry package called Powerview was used. There was an option of using VHDL which would probably have been easier but because many of the ideas implemented were new a more graphical approach was favoured. As many hours were spent running simulations of the processor the graphical approach gave much better visibility of the instruction flowing through the pipeline.

1.5 FPGAs

The idea of the project was not to implement the processor only in simulation but to run it on real hardware. An FPGA (Field Programmable Gate Array) chip allows a logic design to be downloaded to it. The target board has a 'Xillinx Virtex XCV300' chip. This allows up to 300,000 gates to be placed on the chip. It also has 64 Kbits of RAM in 16 blocks called select RAMs. As gates are made from look up tables (LUTs) it is possible use the LUTs as 32 bit RAM cells called RAM blocks. These RAM cells use as much logic one latch.

1.6 Source Material

The information about MIPS microprocessors was taken from a book by Gerry Kane and Joe Henrich called "MIPS RISC Architecture". This book stated all information needed to create an R3000 microprocessor. The book was written for people writing software and so gave very little architectural detail.

2 MIPS Microprocessor Specifications

2.1 Instructions

All instructions are 32 bit and come in three formats (R-type, I-type and J-type). MIPS instructions are three address operations taking two sources and one destination. The

R-type (sometimes called Special) instructions allow a range of register to register operations. The I-type instructions allow a 16 bit immediate to replace one of the operands. The I-type instruction format is also used for memory accesses and for conditional branches. The J-type format has a 26 bit immediate field and the only instruction to use this format is a jump which places the value in the bottom 26 bits of the program counter. A more detailed description of the instruction set is shown in appendix A.

FIGURE 4. Instruction encodings

31... ...0

R-Type	RS	RT	RD	SA	Operation
I-Type	RS	RT	16 bit Immediate		
J-Type	26 bit Immediate				

2.2 Registers

A MIPS microprocessor has 32 addressable registers. Register zero (R0) is special as it is always equal to zero and writes to it are ignored. R31 is a normal register but when executing any branch or jump with store return address, the next PC is stored in R31. In addition to the addressable registers there are three more implemented registers. The Program Counter (PC) is not a part of the main register bank. It is accessible directly through Jump to Register (JR) for writing and Branch And Link (BAL) for reading. The other two registers are LO and HI. These registers are used for the results of the multiplier and divider. Although these can also be also accessed directly by Move To and From LO and HI instructions. All these registers are 32 bits wide although the bottom two bits of the PC should always be zero.

2.3 Conditions

There are no condition flags but instead all branches are conditional on the values of the registers in the main register bank. Each conditional branch instruction specifies two registers (RS and RT) to be fetched and tested. A branch is conditional on the results of two tests. The first is compare the two registers together to test whether they are equal (RS=RT). The other test is simply to look at the sign value (bit 31) of the first register (RS<0). By choosing the second register to be R0 (RT=0) its becomes possible to test RS for less than greater or equal to zero or any combination of the three. For an unconditional branch the Branch if Greater or Equal to Zero instruction (BGEZ) is used with R0 as an operand. This condition will allays be true.

TABLE 1. Branch Condition Table

Branch Condition	Equality Test Required Result	Sign Test Required Result
BEQ	1	X
BNE	0	X
BLTZ	X	1
BGEZ	X	0

TABLE 1. Branch Condition Table

Branch Condition	Equality Test Required Result		Sign Test Required Result
BLEZ	1	OR	1
BGTZ	0	OR	0

2.4 Memory

Memory access instructions are included in the I-type format. The source register (RS) is added to the immediate to create an effective address which is used to reference the memory. The second register (RT) is either used as the destination in a memory load or as a source in a memory store. The memory is byte addressed but is 32 bit wide so all word loads and stores have to be word aligned. Half word accesses have to be aligned to half word boundaries. To help with unaligned loads and stores there are two more memory access instructions. Load Word Left (LWL) and Load Word Right (LWR) in combination allow word loads from unaligned addresses.

2.5 Pipeline Interlocking

MIPS stands for 'Microprocessor without Interlocking Pipeline Stages'. In the MIPS microprocessor this means that some instructions have an implicit delay before their effect takes place. (This is not strictly true as the multiplier/divider has interlocking) The general philosophy is to construct the hardware as simply as possible and, if a result is not ready for use in the next instruction then not to stop the whole processor but use the software to insert instructions into the space. The two main delays in the MIPS microprocessor are branch shadows and memory load delays. There are others but they happen very rarely and will be explained later.

2.5.1 Branch shadow

When a branch is executed the PC is only updated at the end of the next instruction. This is because the MIPS microprocessor designers were using a pipeline that loaded the next instruction from memory while decoding the current. By the time the current instruction is decoded and the CPU detects it as a branch the next instruction is already loaded. The PC is updated by the time the next instruction after that is loaded. The branch shadow is filled with a useful instruction that the branch is not dependent on. If this instruction can not be found then a NOP (Do nothing) instruction is placed to fill the entry. Figure 5 on page 5 shows firstly the compiler unconditionally inserting a NOP instruction into the branch shadow. Then later the NOP is replaced with an instruction not related to the branch.

FIGURE 5. Assembly example

```
ADD  R3, R3, R3
ADD  R4, R5, R4
BGEZ R4, label   <- Branch instruction
NOP              <- Branch Shadow
...
```

```
ADD  R4, R5, R4
BGEZ R4, label   <- Branch instruction
ADD  R3, R3, R3  <- Branch Shadow
...
```

2.5.2 Load Delay

Before a load can complete the address must be calculated and then the load from memory can begin. As this uses two cycles the result is not ready for the next instruction to use as at the time it wants the value the instruction has only calculated the address it is about to access. Again there is a empty entry into which a useful instruction can be inserted if possible. Figure 6 on page 6 shows the compiler unconditionally inserting a NOP between the LW and the next instruction. The NOP is removed if the next instruction does not use the result of the load or an instruction is moved into this space from somewhere else in the code if it is unconditional of the load.

FIGURE 6. Assembly example

```
ADD  R2, R3, R3
ADD  R5, R4, R20
LW   R1, 32(R5)
NOP
ADD  R1, R1, R2
...

ADD  R5, R4, R20
LW   R1, 32(R5)
ADD  R2, R3, R3
ADD  R1, R1, R2
...
```

3 MIPS Microprocessor Construction

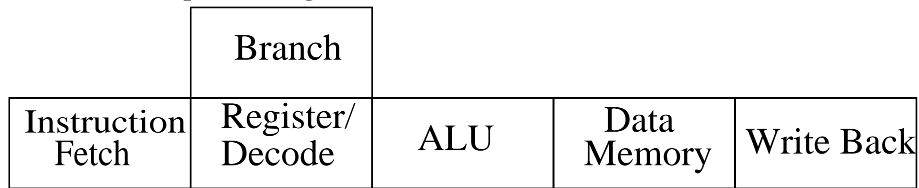
It was decided to create a simple version of the processor which can then be used as a base for the fully implemented version. The smaller version of the processor is called ‘Little Star’. To allow extra features to be placed onto this processor later it is important to make the base with the later components in mind.

3.1 Pipeline

It was an aim to reproduce the processor as it was designed originally and the correct pipeline is essential in order to get the same instruction delay properties as the original without using interlocking. The hints as to the construction of the pipeline come from the non-interlocking properties shown above. Firstly examining the fact that consecutive ALU instructions have no delay means that some form of forwarding is probably taking place. The memory loads have to take the result from the ALU and then pass the calculated address to the memory. The result is ready for use on the next cycle so again a forwarding scheme must be used. The branch shadow on PC altering instructions means that instruction prefetch happens irrespective of the instructions executed. The fact that branch shadow is only one cycle deep means that the PC must be updated within of one cycle of the instruction entering the CPU. By using these rules it is possi-

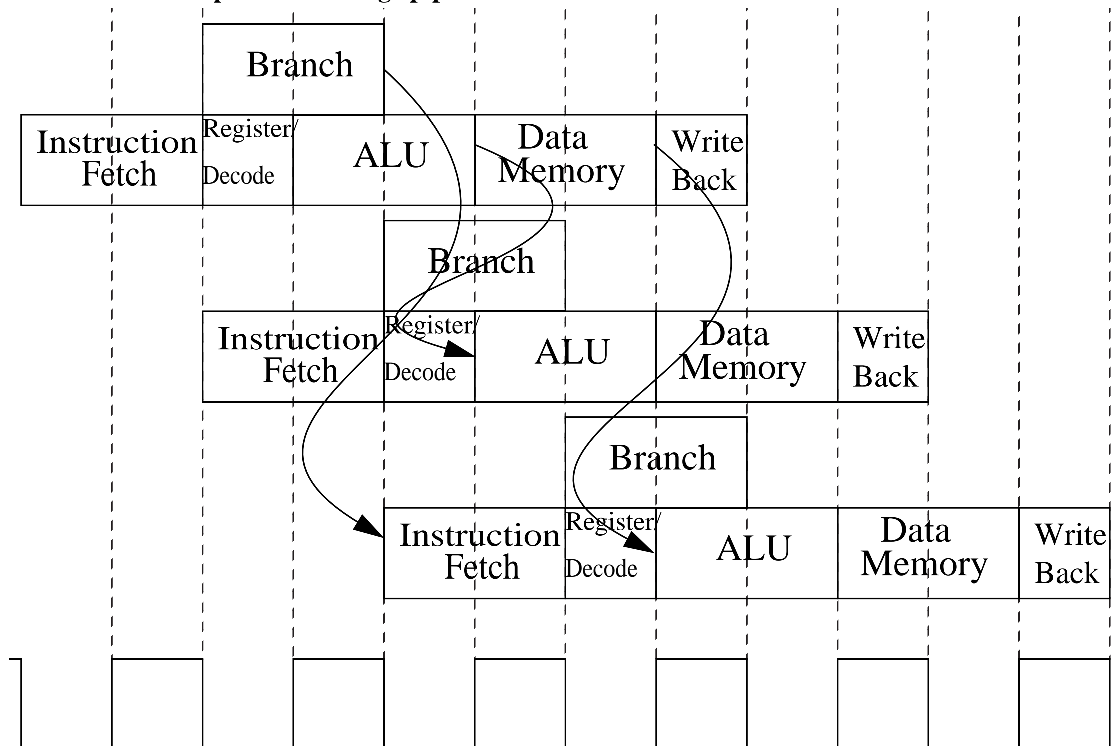
ble to construct a simple pipeline that fits these requirements (Figure 7 on page 7).

FIGURE 7. Simple five stage



There is a problem with this pipeline. The branch unit requires results of the tests on values of registers. Although it is possible to read the registers and test if they match the branch conditions within the decode cycle they will be the values from the register bank rather than the data from the forwarding paths. If the code affected the register in one of the three instructions previous to the branch conditional on this register then the register bank would not be updated. To deal with this the Decode stage is cut down to half a cycle. This way it is possible for the branch to complete within one cycle of the instruction fetch and still get forwarded values only available at the beginning of the ALU stage.

FIGURE 8. Improved five stage pipeline



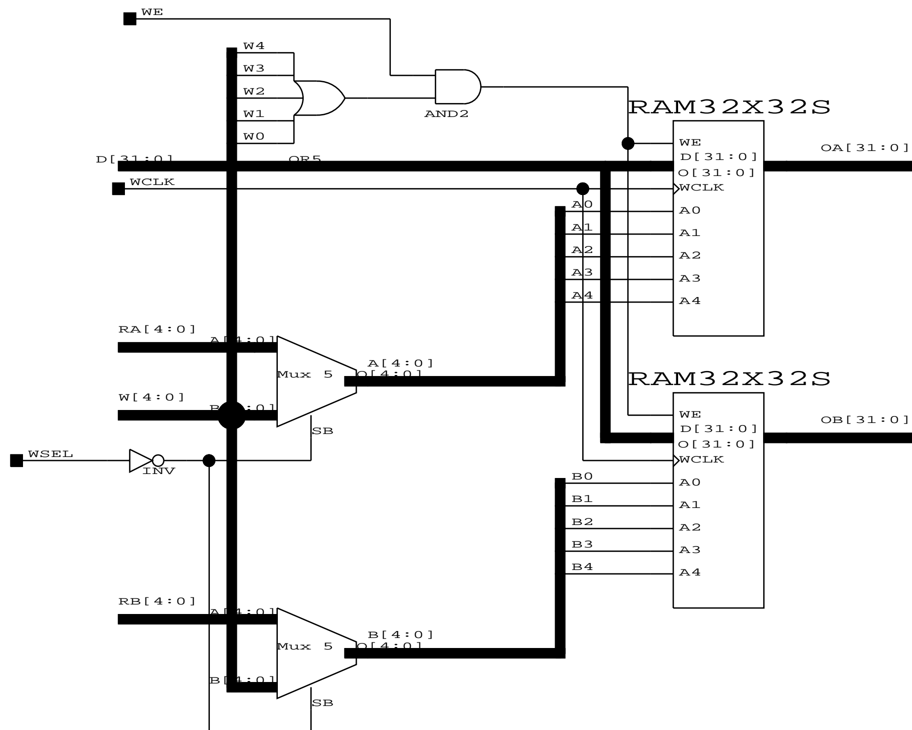
By also decreasing the write back stage down to half a cycle it is now possible to use a register bank that does not need to have the ability of loading and storing simultaneously.

3.2 Register Bank

The register bank is implemented using RAM blocks rather than explicit flip-flops (Figure 9 on page 8). RAM blocks are single ported so two identical copies are required to provide the two simultaneous register reads. Despite this they give a more compact solution than explicit registers. Read and write access is achieved by dividing

each clock cycle into two phases. In the write back half of a cycle the register address to write to is sent to both RAM blocks. The data is written on the clock edge. To ensure data is never written to register zero the write enable line is de-selected if the write address is zero. The read portion of the cycle selects the address to be supplied to the RAM blocks to be the register addresses from the current instruction. These are then read for the next half cycle. After reading, the data is latched outside the register bank. The RAM blocks are explicitly preset to zero so R0 is never written to and will allays be equal to zero.

FIGURE 9. Register Bank schematic



3.3 ALU

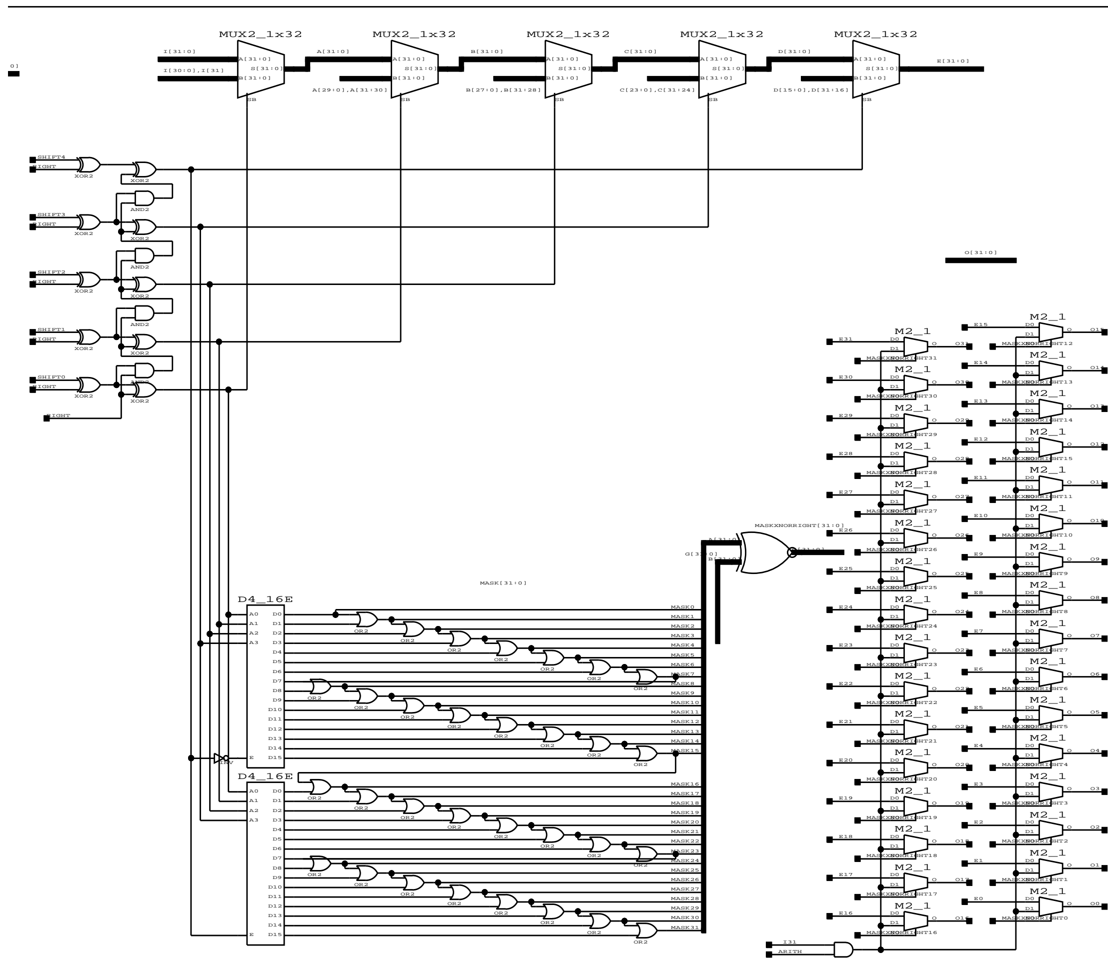
The ALU is made from a logical unit and an arithmetic unit. The results from these are then multiplexed and the desired value is selected. The arithmetic unit is little more than a 32 bit adder with a switchable negator on the second input. The arithmetic unit must also detect overflows. The adder is constructed from cells that contain fast carry logic elements which are especially constructed for carry propagation to increase the speed. Results of tests by Xilinx who design the FPGA chips show that 32 bit additions can be done at speeds of over 200MHz when using fast carry logic units. The logical unit was made with simplicity in mind rather than compactness or speed. Each pair of bits is applied to all four logical operators (AND, OR, XOR, NOR) and the requested one is multiplexed out with a four to one multiplexer.

3.4 Shifter

The construction of the shifter was quite complex because conventional barrel shifter designs require a 32 by 32 grid of elements similar to a 32 input multiplexer. This design is quite fast when designing with custom components but very large when con-

sidered for a FPGA chip. To get round this problem a funnel-type shifter was used. Firstly the value to shift by is negated if the shift is to the right. Then the value to be shifted goes through a series of 5 multiplexers each rotating by a different amount (1,2,4,8,16). Because the negator resolves the least significant bit of the ‘shift by value’ first the least significant rotate multiplexer must be the first in the series in order to get a increase in speed. The series of multiplexers rotate the number to the correct position and a mask must be used to cut out the unwanted part of the number. Left shifts mask out the bottom bits and right shifts mask out the top bits. To make the design simple a 32 way demultiplexer selected with the ‘shift by value’ is used to signal the top bit of the number for right shifts or the top bit to be discarded for left shifts. A series of OR gates propagates the signal to all bits below it. This mask only works for left shifts and has to pass through XOR gates to optionally invert it for use with right shifts. The mask feeds select lines on a block of two to one multiplexers which select between the rotated number and the fill bit. The fill bit is zero for all logical shifts but for the right arithmetic shift it is equal to bit 31 of the pre-rotated input number. The resulting unit allows right and left arithmetic and logical shifts.

FIGURE 10. Shifter Schematic



3.5 Memory

The memory address is calculated by taking a register and adding the 16 bit offset immediate just like an I-type ADD instruction. The other register is not used at this point and is passed to a latch for use in the memory stage even if the instruction is a

load. Passed to the memory stage is the source or destination register and the address to access. Firstly looking at a store instruction as the memory is 32 bit wide and there are no byte select lines on the on board RAM a whole word has to be stored at a time. All store instructions with the exception of SW need to store only a part of the whole word. This means that the word has to be loaded from memory, amended and written back. The MIPS microprocessor supports byte, half and word accesses. In addition it also supports word left and right instructions which rotate the value to be written to memory and mask it into the lower or upper part of the word. Memory interface instructions are described in more detailed in appendix A. To allow all the different memory accesses for the first half of the cycle the memory word is loaded into a 'Memory before write' latch. The source register data is rotated dependant on the memory store type and the two lowest bits of the address. The data to write to memory is then a combination of bytes of the 'Memory before write' and the rotated register data. The memory loads are done in a similar way. There is no need to load the 'Memory before write' latch. This time the data from memory is rotated and then bytes are selected from this and the destination register. To allow byte and half loads with a possible sign extension a method of setting any byte to one or zero is used. The MIPS microprocessor is a Harvard architecture design so it needs to access instruction and data memory simultaneously. In the 'Little Star' this is achieved by taking advantage of the fact that the instruction fetch and data access start half a clock cycle apart. The memory swaps between data and instruction accesses with each phase of the clock. Mentioned before is the method of storing by loading for half the cycle. To allow this two clocks must be used. The general chip clock that drives the majority of the chip is in turn driven by an input clock that is twice the frequency of the general clock. This input clock is used to get a half partition of the half cycle devoted to the data store. The 'Little Star' must run very slowly as a half clock must be longer than two memory accesses.

3.6 Branch

Each cycle the Program Counter updates to one of three values. The first possible value is PC+4. This is the most common case and the PC+4 value is generated using an incrementer. The second possibility is a JR instruction. This feeds a new number to the PC from a general register. It is important to pass the number from the forwarding paths if a newer valid value exists rather than directly from the register bank as the value there might not be updated yet. The third option is a branch, to execute a conditional branch firstly the conditions must be met. The two test registers are taken from the forwarding paths and compared if equal. This result and the sign of the first register are passed to the branch logic. The branch logic returns a flag to the multiplexer stating whether to load the branch target or the PC+4 value. The branch target is calculated by adding the current PC to the 16 bit sign extended immediate. There is one more case that is not covered and that is the jump. The jump takes a 26 bit immediate and places it in the bottom of the PC preserving the top four bits. This encoding was done by reusing the branch adder to save space. The bottom 26 bits from the PC are nulled leaving only the top four bits from the old PC value to be passed to the adder. The immediate is passed as a 26 bit rather than a 16 bit value. To record the return address the PC+4 value is latched and then multiplexed onto the pipeline as the result of the ALU stage.

FIGURE 11. Branch Schematic

3.7 Forwarding

The destination register number is usually explicitly specified in the instruction. Instructions that branch and link the return address need to have the destination register set to R31. All other instructions that do not have a destination have the destination register number set to 0. Each instruction unconditionally fetches the two registers specified in the instruction from the register bank. The destination register number is passed down with the instruction along the pipeline until it reaches the write back stage. The requested source register number is compared to each of the destination register numbers flowing through the pipeline. If any of them match then the register bank value is discarded and instead the value from the forwarding path is multiplexed in. It is important that the value forwarding from the ALU stage takes priority over the one from the memory stage so as to get the more recent value. The comparators that detect if register numbers are equal also make sure that R0 is never forwarded as any calculated value is meant to be discarded and may be not equal to zero.

4 Advanced MIPS specifications

All these elements in combination make a processor capable of running MIPS code. This simple version processor was called “Little Star”. At this point in the construction a lot of testing took place to verify the underlying structure of the processor. Although ‘Little Star’ is capable of running code it does not implement some of the ‘advanced’ features of the original chip. The full R3000 microprocessor compatible chip is called ‘Yellow Star’ and includes a cache, memory management, coprocessors and exact interrupt handeling. These additional features are described below.

4.1 Cache

The original MIPS microprocessor had two caches: an instruction cache and a data cache. Both are transparent to the user. The original design had four kilobyte direct mapped caches. The caches are one word (four bytes) wide and mapped using the next 10 bits of the address. The cache outputs the data of the selected entry and a 20bit value tag that is possibly the top of the physical address. As the caches are transparent to the user they can be changed for different sizes and types. Some regions of virtual memory are non cachable and this overrides he cache hit flag and cache writes.

4.2 Memory Management

The memory management on the MIPS microprocessor constitutes of a 64 entry TLB (Translation Look-aside Buffer). This TLB is filled using software. The same TLB is used for both instruction and data accesses but the 'Yellow Star' pipeline allows this as the instruction and data accesses are offset by half a clock cycle. This means that a TLB lookup is only half a cycle long. The TLB is fully associative with 4 Kbyte pages. This leaves 20 bits that are used for lookup. If the 20 bits passed to the TLB from a virtual address match an entry then the 20 bits of the physical address of the corresponding entry are output and the hit line is raised. The 20 bits from the TLB are combined with the bottom 12 bits of the virtual address to create a 32 bit physical address. The entries in the TLB are modified by software. The programmer can enter the two 20 bit translation fields (virtual and physical) along with a region of other information as a 64 bit value into any entry in the TLB. There is a six bit ASID (Addressable Space ID) field to store the process ID number that the page is meant for. This allows the kernel to swap processes without flushing every entry in the TLB. The other flags in the entry are: Noncachable, Dirty (an exception is caused if trying to write to a non-dirty page), Global (allows any process to access the page) and Valid. On a MMU exception the 64 bit entry that matched along with the virtual address are written to special registers for the kernel to decide which page caused the exception and which to insert or amend.

4.3 Coprocessors

The MIPS microprocessor has an interface to handle up to four coprocessors. Coprocessors can have 32 general registers and 32 control registers although not necessarily all are used. There are four R-type instructions that allow transfers of registers between the general register bank and coprocessor general or control registers. These instructions have the same delay as a memory load instruction. Although not stated in the sources all evidence points to the fact that the interface with the coprocessors happens in the memory stage. There are memory load and store instructions that load and store directly coprocessor general registers to memory. These instructions use the processor main registers as an address base but use the coprocessor registers as a source or destination. Each external coprocessor has a flag line connected with the CPU that can be tested and a conditional branch executed dependent on its value. Coprocessor instructions can be executed directly from the instruction stream. A 25 bit field specifies the instruction while two of the remaining bits report the coprocessor number.

4.4 Coprocessor Zero (CP0)

Coprocessor Zero is an internal coprocessor which is used for controlling memory management, exceptions and some other options. CP0 has a selection of registers which are used for controlling the behaviour of the processor.

4.4.1 CP0 Interrupt

The status register is one of the more important registers. The register has several fields. The current Kernel/User (KUc) flag states whether the CPU is in kernel mode. The current Interrupt Enable (IEc) flag states whether external interrupts are turned on. If cleared then external interrupts are ignored until the flag is set again. In an exception these flags are copied to previous Kernel/User and Interrupt Enable (KUp and IEp) and then cleared so the system moves to a kernel mode with external interrupts disabled. The Return From Exception instruction writes the previous flags to the current flags. All these flags are kept in the status register. The Interrupt Mask (IM) field has eight flags to individually disable any of the external interrupts by clearing any of the flags. The BEV flag controls the exception vector.

TABLE 2. Exception Vectors

Exception	BEV=0	BEV=1
Reset	-	0xbfc0 0000
TLB Refill	0x8000 0000	0xbfc0 0100
Multiply/Divide*	0xbfc0 0300	0xbfc0 0300
Other	0x8000 0080	0xbfc0 0180

The SU four bit entry controls the usability of each of the four coprocessors. If a bit is set then the corresponding coprocessor is usable and does not cause a coprocessor unusable exception when accessed. If the CPU is in kernel mode CP0 is usable even if the SU bit zero is not set. The Cause register has fields of what the cause of the exception was. The five bit Exception Code states the exception number.

TABLE 3. Exception Codes

Code Number	Exception Name
0	External Interrupt
1	TLB modification
2	TLB exception on a load or instruction fetch
3	TLB exception on a store
4	Address error on a load or instruction fetch
5	Address error on a store
8	Syscall
9	Break
10	Reserved instruction
11	Coprocessor Unusable
12	Arithmetic Overflow
13	Multiply/Divide (‘Yellow Star SM’ only)

The eight bit Interrupt Pending field indicates which interrupts are pending. The top six bits are connected to the six external interrupt lines. The bottom two bits are writable and allow the software to cause software interrupts. The two bit Coprocessor Error field states which coprocessor was attempted to be accessed at the time of the Coprocessor Unusable exception. The Branch Delay (BD) flag states if the exception was caused by an instruction in a branch delay. The EPC register holds the address of the instruction that caused the exception unless the BD flag in the Cause register is set in which case the causing instruction is the next instruction. On any TLB or memory exception the Bad virtual address register stores the address that caused the exception.

4.4.2 CP0 Memory Management

Two registers (EntryHI and EntryLO) hold the 64 bit data value for transfers with the TLB. EntryHI holds the Virtual Page Number and the ASID value. EntryLO holds the Page Frame Number along with the four flags (Noncachable, Dirty, Valid, Global). To access TLB values there are four instructions: a TLB Probe instruction probes the TLB for an entry that matches the Virtual Page Number in EntryHI, The matching TLB entry is the loaded into EntryLO and EntryHI, a TLB Read instruction reads the entry pointed to by a the CP0 Index register. The Index register is simply a six bit field that is used to index the TLB. The TLB Write Index instruction writes the contents of EntryLO and EntryHI registers to the TLB entry addressed by the Index register. The TLB Write Random instruction writes to the TLB entry addressed by the Random register. The Random register has a six bit field that is incremented every cycle. The Random register never drops below eight to allow eight safe entries that will not get overwritten by the operating system.

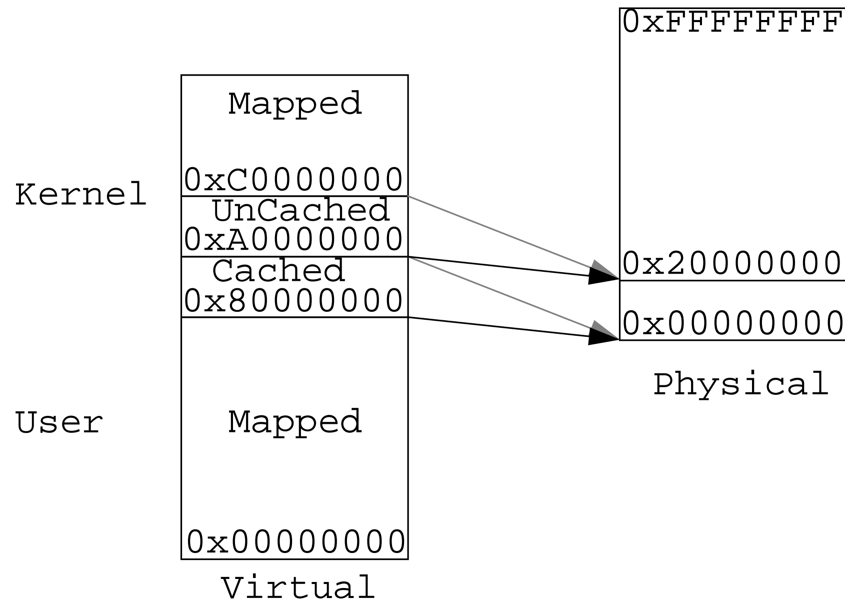
4.5 Exceptions

The behaviour of the MIPS microprocessor during an exception is to store the PC of the instruction that faulted in a coprocessor register and to make sure that it and none of the already fetched instructions succeeding it are executed. For most instructions this stops the result value from being written to the register bank. The PC is changed to the interrupt vector. The processor then continues from the interrupt vector running exception handling code.

4.6 Memory Map

The virtual memory is divided into kernel and user spaces. The bottom two gigabytes is the user space whose addresses are mapped through the TLB. The top two gigabytes is kernel space divided into three further parts. The first half gigabyte of kernel space is unmapped and cached. The second half gigabyte of kernel space is unmapped and uncached. For the two unmapped sections the virtual address has its top three bits cleared so as to map the virtual sections to start at physical address zero. The remaining gigabyte of kernel virtual space is mapped. An access from a non-kernel mode to any kernel space results in an address error exception.

FIGURE 12. Map of virtual and physical memory



4.7 Multiplier Divider

The Multiplier Divider unit takes R-type instructions that designate two registers to be acted upon. The results are written to two registers (HI and LO) with no delay. These registers can be read to the register bank. This unit is interlocked and may cause the whole processor to pause.

5 Advanced MIPS Microprocessor Construction

Taking 'Little Star' as a base the extra features can be added around it to create 'Yellow Star'. Unfortunately these features are heavily interlinked and so have to be added all at the same time.

5.1 Cache

The two caches run independently but share the physical RAM due to design having separate data and instruction memory ports. To share the RAM in the 'Little Star' it was possible to swap resource allocation by clock phases. In the 'Yellow Star' a memory access is a lot longer than half a clock cycle as it runs from cache which is faster than memory. Firstly to solve the sharing problem two cache busses are constructed to allow the Harvord architecture to access both data and instruction caches at the same time.

If a cache miss occurs the whole processor is paused for several cycles until the data or instruction is loaded and the cache is refilled from the external memory. If the access is a write then during the pause the instruction cache is supplied with the address being written to. If the instruction cache signals a hit then the value in the cache must be updated before the processor is released from the pause. This achieves instruction cache coherency

FIGURE 13. Cache bus structure

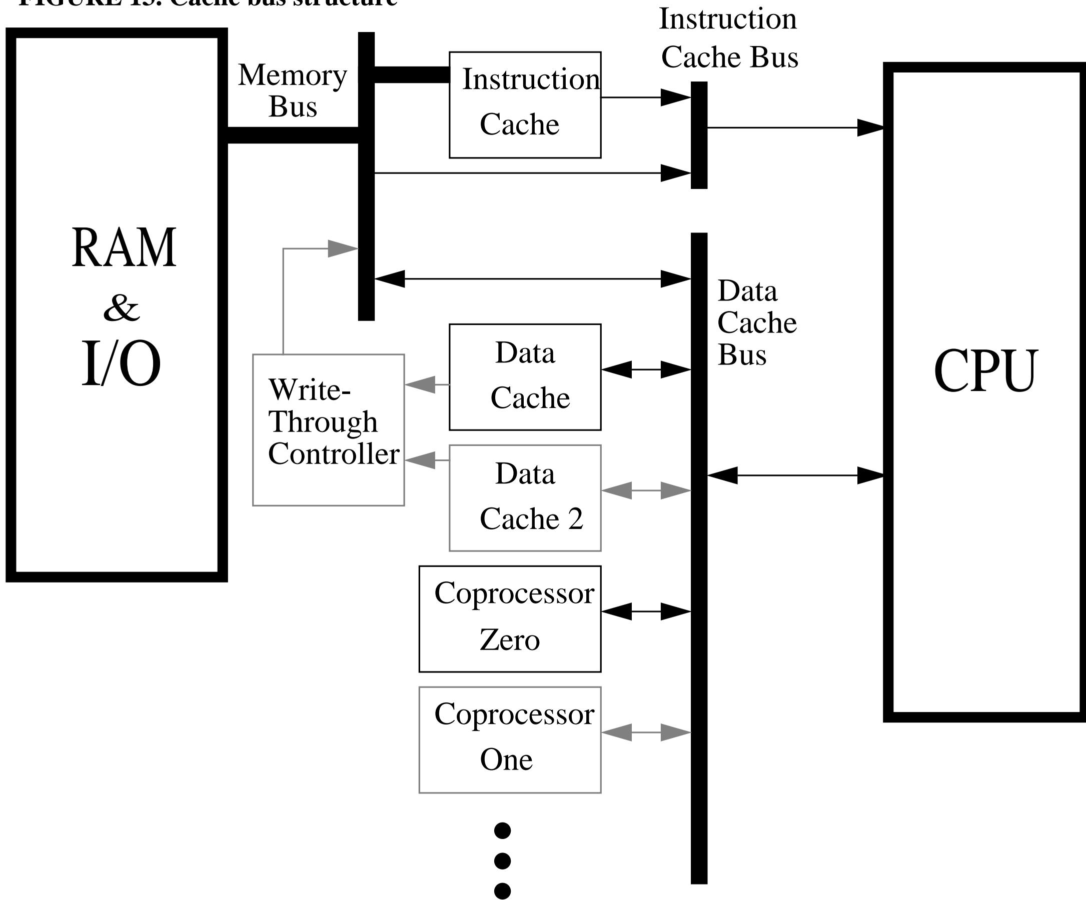
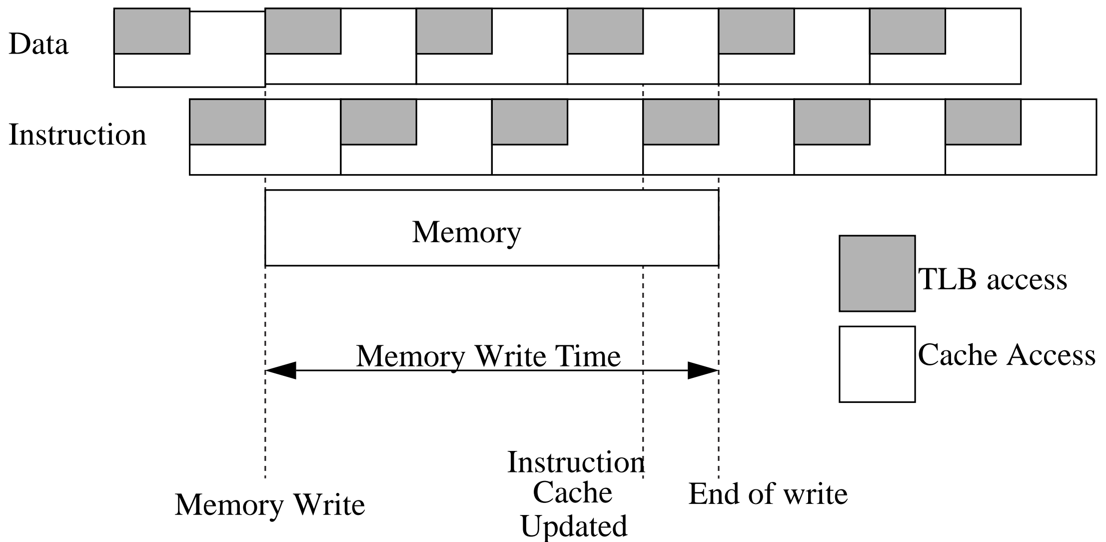


FIGURE 14. Instruction cache update



As the cache in the original design was direct mapped this made the design very simple. It can simply be made out of RAM. The biggest RAM elements on the FPGA are the select RAMs. All the select RAMs combined give a total of 64 Kbits of RAM. All that needs to be stored in the cache is the 32 bit RAM value, the 20 bit physical address and a valid bit. This makes up to a 53 bit field. Unfortunately multiplied by the 1024 entries this gives 53 Kbits of RAM used per cache. So there is not enough space on the FPGA to fit both the caches. With the total of 64 Kbits of RAM space it is possible to

create 512 entry caches instead as they are transparent to the user. The 32 bit RAM value, the now 21 bit physical address and the valid bit now make a 54 bit field reproduced 512 times. This uses 27 Kbits per cache so for both the instruction and the data cache only 54 Kbits of select RAMs are used. The two 2 Kbyte caches should give a performance not much lower than that of the R3000 microprocessor. Other cache combinations are possible using the 64 Kbits given.

TABLE 4. Direct Mapped Caches

Entries	Field(bits)	RAM(Kbits)
1024	53	543
512	54	27
256	55	13.75
128	56	7

TABLE 5. Possible direct mapped cache combinations

Cache1 entries	Cache2 entries	RAM
1024	128	60Kbits
512	512	54Kbits

The bus structure in Figure 13 on page 16 shows the cache bus attached to the memory bus through bidirectional tristated link and not through the cache. The cache instead writes and reads from the cache bus. This might seem quite strange but the reason behind it is to allow more than one cache to sit on the cache bus. A victim cache that caches values discarded by the main cache can be placed along side the main cache. This also allows the use of write back caches rather than write through. This gives many more possibilities.

TABLE 6. Multi Associative Caches

Entries	Field(bits)	RAM(Kbits)
1024	54	54
512	55	27.5
256	56	14
128	57	7.125

A large range of combinations becomes possible. For example the instruction cache having a 512 entry direct mapped cache (27 Kbits), the data cache having two 256 entry multiassoceative caches (24 Kbits) and one 128 entry victim cache (7 Kbits). As long as the caches have a protocol and a priority order then any number of caches of any type and size can be used. There is a limit of using no more than 12 bits (4 Kbytes) from the bottom of the address for lookup as this is the portion of the address that is not affected by the memory management. To get around this limit multi-associative caches can be used to break the 4 Kbyte barrier.

5.2 Memory Management

The TLB in the R3000 microprocessor was constructed from a block of CAM (Contest Addressable Memory). Although this would be the most logical method this would require a tristate buffer for each of the 64 bits in the 64 entries giving a total of 4096 tristate buffers. The FPGA does have that many tristate buffers but this would make the TLB take up over half the design space. Instead on 'Yellow Star' the TLBs is arranged into a block eight by eight. Each element is only used for matching the incoming virtual address. If an element hits (matches the incoming entry) then it discharges two tristated lines, one going down, the other going across the array. The address of the matching entry is then reconstituted into a six bit address. The six bit address is used to lookup in a RAM block to get the actual 64 bit entry values. This entry is passed out for use by either the memory address translation or TLB read instructions explained later. The matching cells only hold the virtual address, the ASID field and the global bit. This data is duplicated in the block RAMs. This is all that is needed to match an address. If lookup address is in the kernel unmapped space then the input address is passed out instead of the one looked up in the block RAM. The unmapped address will have its top 3 bits cleared so they point to physical address starting at zero. If the uncached bit is set in the hit page then the cache is ignored and a memory access is forced. If the dirty bit is not set on a write an exception is triggered.

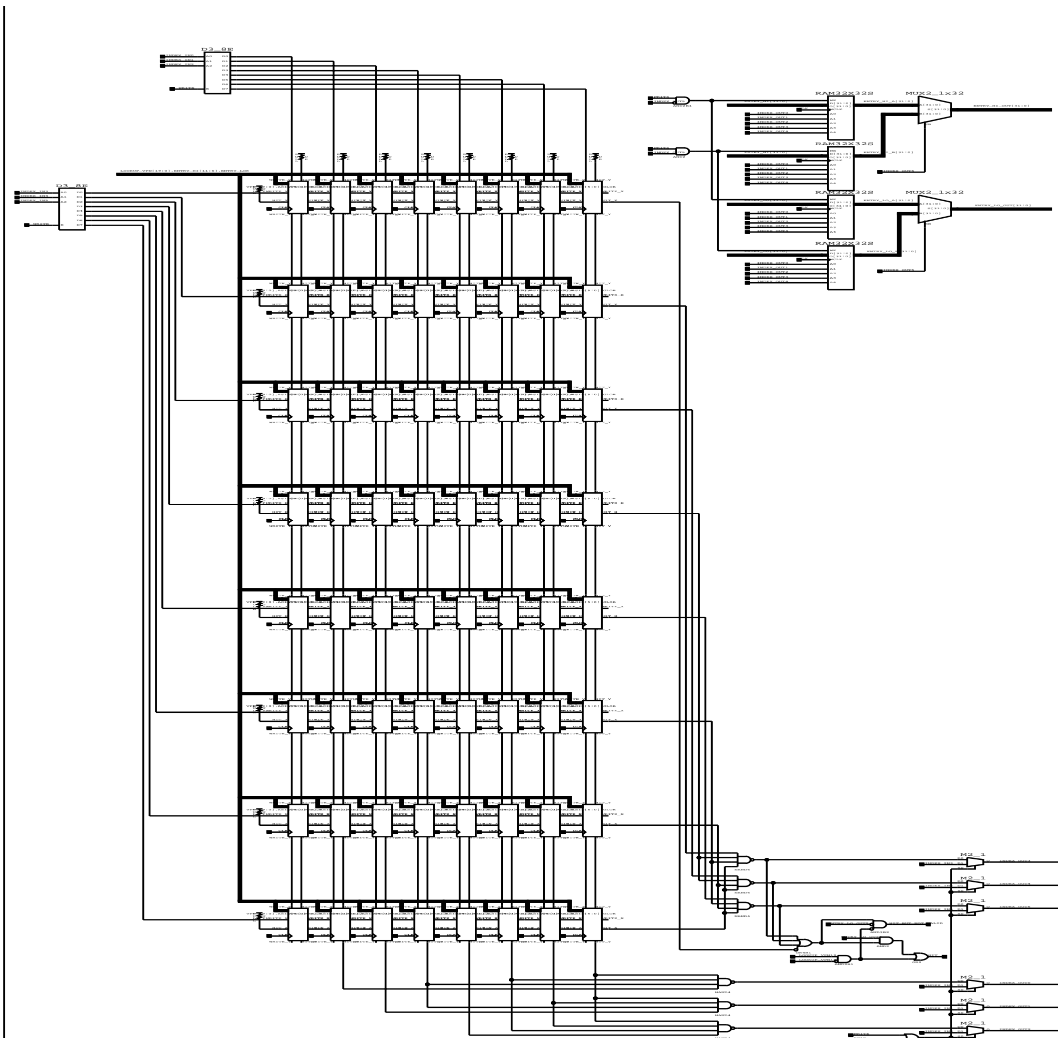
5.3 Coprocessors

The coprocessors have to link with the central processor to transfer data and to pick up instructions. The only way the coprocessors can get instructions from the instruction stream is to snoop on the instruction cache bus. Similarly the coprocessors need to load and store memory and so need to be attached to the data cache bus. This theory is reinforced by the fact that transfers between the coprocessor and the CPU have a delay of one cycle and so must happen in the memory stage. The coprocessors get their instructions by snooping the instruction cache bus. If there is a load from coprocessor instruction or a store coprocessor register to memory instruction then the coprocessor waits two cycles before driving the cache bus with the requested register value. Similarly if the a load from memory or store to coprocessor instruction the coprocessor waits two cycles before catching the data and sending it to the correct register. It is important that the coprocessors monitor the halt line from the main CPU so as to not to become unsynchronized and take control of the data cache bus at the wrong point. Also the coprocessors have to watch the exception flush line from the CPU to make sure if to commit to instructions.

5.4 Coprocessor Zero

Coprocessor zero was interfaced into the processor just like all other coprocessors even though it is special as it has many other connections to the CPU. The coprocessor zero memory management instructions are executed in the memory stage where the data access TLB lookup gets replaced with the CP0 TLB operation. As an instruction can never be a CP0 instruction and a memory access it is perfectly legal to take control of the TLB for this cycle. Most of the registers are writable but also get written when the processor enters an exception. This is solved by multiplexing the write values making sure the exception has priority when writing. Most of the construction of CP0 is

FIGURE 15. Memory Management Unit Schematic



described in the MMU and exceptions sections. (Section 5.2 on page 18 and Section 5.5 on page 19)

5.5 Exceptions

If an instruction causes an interrupt the processor carries on until the instruction is at the end of the memory stage. This is the last point where the instruction can cause an exception but can also be aborted and the processor flushed of all instructions. The multiplexer in front of the PC is switched to load the interrupt vector value into the PC (Figure 11 on page 11). All instructions flowing through the CPU have a copy of the address where loaded from. This address is taken from the PC at the time of a load and passed from latch to latch in each pipeline stage. If the preceding instruction is a branch that is executed the carried instruction PC value is not updated as the instruction is in a branch shadow. This allows the kernel to return to the code and re-execute the instruction. If the instruction causes an interrupt the EPC (Exception Program Counter) in the coprocessor is loaded with the carried instruction PC value. The processor enters the kernel mode by clearing the user bit and disallows the interrupts by clearing the interrupt enable flag. Before being cleared these two bits are stored in the previous user mode and interrupt enable flags. In turn the ‘previous’ flags are stored in the old user

mode and interrupt enable flags. If the exception was a memory management exception then the accessed virtual address is stored in the FaultAddr register and the EntryHi and EntryLo registers are filled with the faulting page entry in the TLB. If the TLB fault was caused in the instruction fetch stage then the forwarded PC address of the faulting instruction is passed onto the datapath in the ALU stage so in the memory stage it recreates the MMU fault for recording. The exception code is carried along with the PC of the instruction. If an instruction causes an exception in the further stages of the pipeline the instruction's carried exception code is checked for the valid flag and if the instruction has not caused an exception yet then its code gets replaced with the code of the exception just detected. This exception code gets written to a CP0 register. By recording these values to CP0 it is possible to recover from any exception.

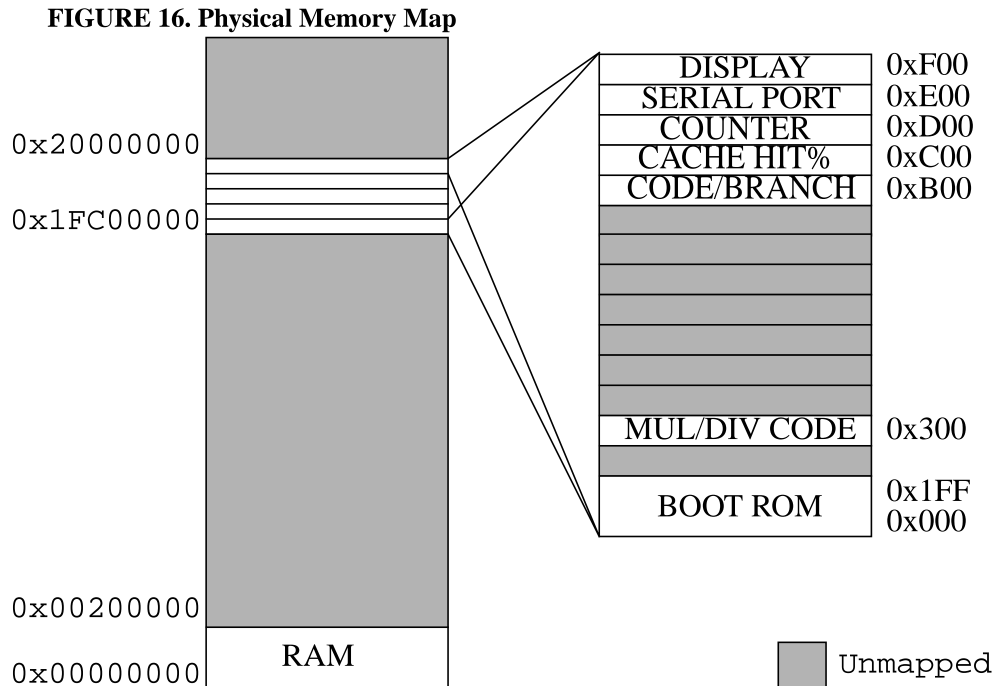
5.6 Memory Map

The processor has to communicate with the outside world and so a serial port was memory mapped. The serial port has two ten entry input and output FIFOs. Reading from the serial port gives: an eight bit value of the entry at the front of the input FIFO, a valid bit of the value, and the output FIFO full bit. There are two addresses where the serial port can be read. The first allows reading of the value and popping it off the stack and the second allows a snoopy read. The input value valid bit is also connected to an external interrupt input line. As the processor starts at the RESET vector the memory at that address has to be a prewritten RAM. It is possible to preset the RAM blocks in the FPGA. A very small program was written in the RAM blocks to take the data from the serial port and write it to RAM and then execute it. Firstly it takes a three words as the program counter to jump to when the program is loaded then, the start address where to write the program and the length of the program in bytes. Then the program stream is sent. The loaded program can be a more sophisticated loader. The board has two buttons and an eight segment bargraph display that are mapped. An extra button was reserved for driving the reset line. There is a down-counter that can be set to a value and every cycle when the processor was not halted it drops by one. When it reaches zero it raises an interrupt line. This down-counter is only activated when the processor is in interrupt enabled mode otherwise it does not count down. The main RAM was mapped at the bottom two megabytes of the physical address space.

5.7 Multiplier and Divider

The multiplier and divider on the R3000 microprocessor have complex interlocking but fit the pipeline. The interface to the multiplier and divider was left empty and the units never implemented as they would have probably been too big for the FPGA. Also the time scale was too tight to implement more large components with full testing. The interface allows the unit to take the values from the forwarding paths or register bank and act on them. The result is multiplexed onto the datapath at the end of the ALU stage on a Move From HI/LO instruction. If the instruction comes before the multiplier/divider has finished the processor is paused until the data is ready. An element makes sure that is the data ready signal comes at any point the processor is only released on the same clock phase as it was halted. This allows an asynchronous multiplier/divider. A flag in Coprocessor zero register switches the processor into a 'Yellow Star SM' (Software Multiplier). In this mode: multiply, divide and access HI/LO register instructions cause an exception. These instructions cause a different exception code.

FIGURE 16. Physical Memory Map



Another flag can switch the exception vector for these exceptions to (0xbfc0 0300) fast Multiplier/Divider code.

6 Debug

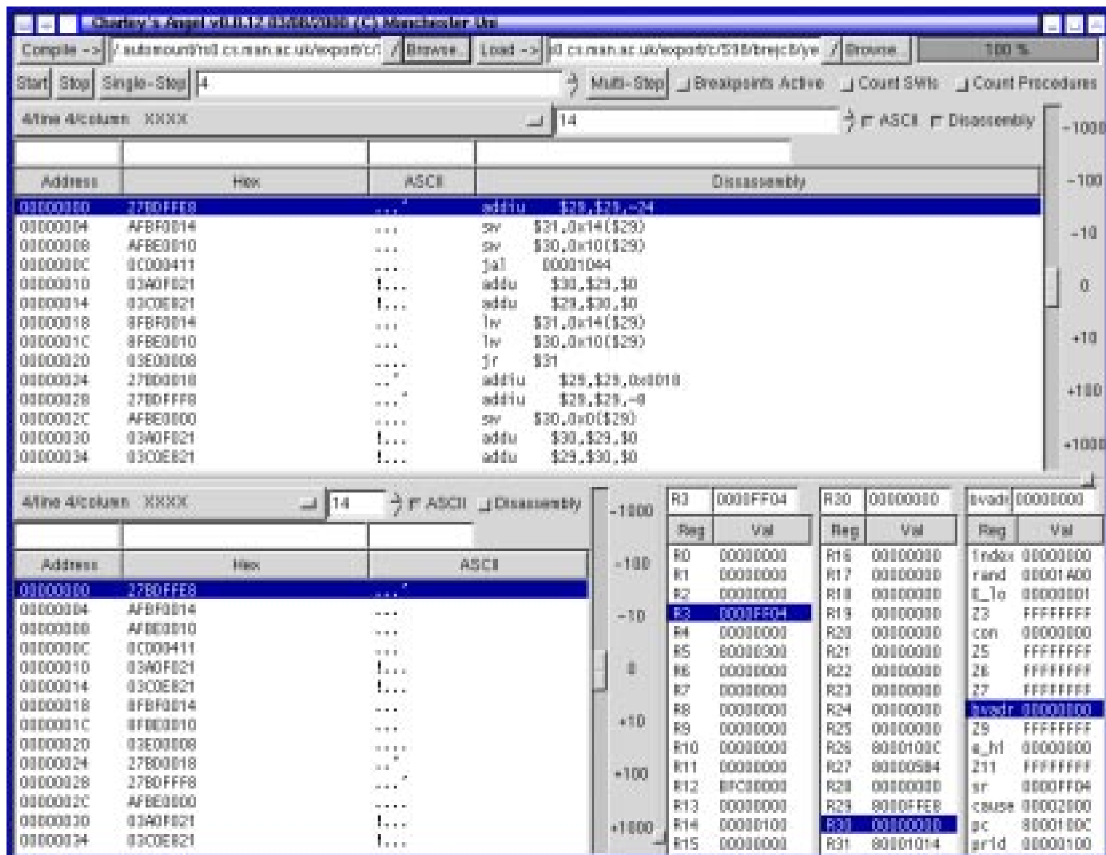
6.1 Binutils and GCC

In order to write reasonable programs it is necessary to make an assembler and a compiler. GNU Binutils and GCC can be compiled to create a cross compiler for a MIPS-unknown-elf. As there are no operating systems on the chip the target does not have any way of input or output or any operating system functions. The compiler and assembler need to be specially set up to create programs that can run on a system with no operating system. A program (Progload) was written to get the three pieces of information from compiled programs and then send them over the serial port. This allowed any program to be compiled and then sent to the board for running.

6.2 Charlie’s Angel

Charlie’s Angel is a front end debugger for communicating with an external board using a serial connection. The board must be running a program to process the commands from the Charlie’s Angel. This program was written and then sent to the board using Progload. The back end responds to a host of instructions from the front end. For stepping or multi-stepping through code the chip uses the down counter to set the number of instructions to take before interrupting back.

FIGURE 17. Screenshot of Charlie's Angel



7 Results

Both 'Little Star' and 'Yellow Star' were tested and debugged and functioned correctly under many with many test programs including the SPIM (MIPS microprocessor software simulator) self test code. A simplified version of 'Little Star' only allowing word loads and stores was tested running code from the on-chip RAM blocks and achieved speeds of 50 MHz correctly executing a range of instructions especially selected to use the longest paths. This version is probably capable of running at higher speeds but the on-board clock was limited to 50 MHz. 'Yellow Star' was tested running memory mapped code from cache without fault. There are units installed in the processor to test parameters like the cache hit rate and periods between branches and output the results collected through memory mapped registers. Although these have been tested and found to be working they still await execution of large complex code for valid results. With the smaller programs executed for testing these units have replied 99.9% cache hit rate as the cache is bigger than the program and all periods between branches being eight or five as this was the loop length of the test program. The 'Little Star' is made from 16,556 gates and could be further compressed while 'Yellow Star' uses 63,327 gates (Table 7 on page 23). These gate counts are derived from the simulator. Although the Virtex claims to be capable of containing up to 300,000 gates a 30,000 gate design uses a quarter of the chip. This is understandable as blocks that the Virtex is made from can rarely be used to their full potential. This means that on a design like this the maximum gate count that can fit onto the Virtex is about 120,000 gates, enough for an array of up to seven 'Little Star' processors. By optimising the design and halving the caches

two ‘Yellow Stars’ can fit onto a Virtex and run in parallel. One ‘Yellow Star’ can emulate another device like a floating point unit.

TABLE 7. Gate count statistics

Component	Gate count
Yellow Star (excluding cache)	63,327
Little Star	16,556
MMU	30,892
ALU	2,102
Shifter	2,220
Register Bank	1,061

8 Conclusion

The project was completed successfully seemed to run all MIPS code correctly although not fully tested yet. This was a very taxing project and a lot was learned from it. One of the more important lessons is that testing does actually take longer than implementation. Even though each unit was tested before being integrated with the processor when competed they still failed in situations never conceived. To create a large design on any platform it is important to use the components that are cheap in that technology. On an FPGA, RAM blocks are very cheap and tristate buffers are costly. To create a small and fast design RAM blocks were used where possible to replace more expensive components. Tristate buffers were avoided and only used where necessary e.g. Data cache bus can be driven by the: processor, memory, cache or coprocessors registers so having a 32 bit wide 20 input multiplexer would not be feasible. Another good example of this is the construction of the TLB where again using a 64 bit wide 64 input multiplexer would have been excessive to avoid tristate buffers. A compromise was reached and worked very well. The same problem arises in the construction of the register bank and the shifter but in these cases the solution was created without the use of tristate buffers by using gates and RAM blocks. This is a common theme that occurs when designing for FPGAs, a result bus has to be driven from a large number of places. There is no one solution to the problem but new solution has to be created every time very often they have similarities with the ones solved here. Greatest lessons were in the ways to approach a problem.

After creating ‘Little Star’ a lot of time was spent trying to understand everything there is to know about exceptions and caching. When designing the exception handling every possible instruction and state was considered before any implementation was done. The first implementation of the exception handling circuits used a multiplexer to read in the exception vector to the PC and a few registers to store the instruction PC. This took a few minutes to implement and more was learned from looking at an exception happening in simulation than over the several days studying it. Inserting test unit into a working project that will give a lot of useful information. Having a modular design with clean interfaces allowed large changes very late in the construction which is what happened several times but luckily without the need to make changes to the interfaces. The processors can be used as a base for testing other units as they allow: compilation of large pieces of code, step by step testing and high performance. Ultimately it is

hoped that these processors could be placed into systems that can easily be placed to monitor or control peripheral devices or test components.

MIPS(R) and R3000(R) are registered trademarks of MIPS Technologies, Inc. in the United States and other countries. Charles Brej is not affiliated in any way with MIPS Technologies, Inc.