Platform Basics

Pete Wilson • Kiva Design • pete@kivadesigngroupe.com
Copyright © Pete Wilson, 2006. All rights reserved

Introduction

We are interested in how to design and deploy software onto multicore designs. Before delving into the intricacies of such platforms, we should revisit current-generation platforms - processor architecture, the I/O and interrupt architecture, and the architecture of an RTOS. In this essay we will describe and explain a canonical system - none of the components exactly reflect real-world product, but are sufficiently close that we can explore how well software works on the platform and then, later, describe a multicore version of the architecture and show how to adapt to the new capabilities of this platform.

We provide an informal overview of the system components and how they work, in two sections - *architecture* (which describes the basic properties of the system components without (much) reference to implementation) and *implementation*, which provides information on actual implementation. For the processor, we will describe several implementations or *microarchitectures*. We look here at the hardware, software is dealt with in a separate essay.

Hardware Architecture

The Processor

Our processor is a simple RISC machine. It has a number of registers, an instruction pointer, the ability to read and write memory in the usual manner, and a normal complement of logical and arithmetic operations. It provides a straightforward page-based protection mechanism (which can also support virtual memory), and the machine operates in either user mode or privileged mode - certain operations are illegal in user mode. Again, this is all terribly normal and pedestrian.

The processor also has a straightforward interrupt scheme. An external device can interrupt the processor by asserting a signal on an interface; at the next instruction fetch, the processor notices the signal and instead of executing the instruction it 'expected' to, it saves some state and branches to a pre-specified address. That address should contain the first instruction of whatever software is needed to handle the interrupt; when all necessary work has been done, the software restores state and execution resumes with the instruction the processor 'expected' to execute.

As with other processor architectures, our machine executes interrupt-handling code in privileged mode; it also contains a register which when set to '1' will stop the processor acting on the presence of an interrupt; this register is set to 1 when an interrupt is recognised.

The protection mechanism is page based - all pages are the same size (4KB) and each page has read, write and execute permissions along with a base address and a physical address. A processor implementation is equipped with hardware which can hold some number - implementation-dependent - of page table descriptors. On every memory access (whether instruction fetch or memory read or write), the processor looks up the address in its set of descriptors. If one of the descriptors holds a base address which specifies a 4KB slice of memory con-

taining the address being looked up, then we have an address match, and the hardware provides the corresponding physical address to the processor along with the permission bits. The processor then checks that what it's trying to do is legal, and if so provides the physical address to the memory system. If there is no match for the address, an interrupt is raised. If the access is illegal, an interrupt is raised.

DMA

An embedded system generally controls some devices; in some systems, the amount of data provided to or by the devices is rather small, and the data movement is most conveniently done by software running on the processor. In others, fairly large amounts of data are moved, and it is inconvenient and inefficient to use software; rather, a specialist helper engine - the DMA - is provided. This is a very simple processor, with a very limited instruction set whose purpose is to move data from devices to memory, from memory to memory, from memory to devices, and between devices.

Our DMA engine is a multi-channel machine; it is capable of running several transfers concurrently. Each transfer is defined by a *channel program*, a set of commands in memory. Each command specifies a transfer, defining source and destination addresses, the stride for source and destination, how much data to transfer, and where the next command is. As is usual, the addresses are physical addresses. When reading from or writing to a register (generally a device FIFO) the stride is zero. The next command is indicated by a pointer, and upon encountering the last command, the DMA will interrupt the processor. The DMA engine multiplexes itself between multiple channel programs at the level of small bursts of transfers (if the transfer is memory-to-memory, the burst will be some number of cache lines in size; if it involves a device, the burst may be smaller than a cache line if the device has a small data buffer). The various channel programs are interleaved on a time-slice priority basis.

Implementation

Processor

We will outline three processor models - two *in-order* machines, one with a short pipeline and one with a longer pipe, and one *out-of-order* machine. The three models provide a fairly wide spread of performance and size (when implemented at equivalent effort in identical technologies). We will later use these machines to demonstrate that the enhancements we suggest for multicore provide benefits at the *architectural* level - that is, across a range of implementations.

K100

The K100 is a 4-stage pipeline, in-order machine optimised for silicon area. Its performance is high in one sense - it will execute around one instruction per clock - but is compromised in frequency by the short pipeline. It is illustrated diagrammatically in Figure 1.

The pipe stages are fairly conventional. A self-incrementing *Instruction Pointer* contains the address of the next group of instructions to fetch; every clock, this address is sent to the memory and the memory provides a value on the next clock. The value, which is some number (1, 2 or 4) of instructions, depending on the variant, is broken up into instructions and inserted into the *Instruction Queue*, a FIFO structure which can hold some number (1, 2, 4 or 8) of instructions. If there's an instruction available in the queue, every clock it is offered to the *Decode and Register Read* stage, which will accept it unless the *Interrupt* box indicates that there's an inter-

rupt which needs to be serviced. In the ordinary case, the Decode and Register Read stage accepts the instruction, decodes it, reads any registers which may be specified (up to three, for a store) and passes the values on to the *Execute* stage. In this pipeline, all instructions, including multiply and floating point ops, execute in one clock - this cramming of logic into one stage saves some area, but forces a stark limitation on clock frequency. The one exception is divide, which does two bits per clock and can therefore take from 4 to 16 clocks. Upon completion of execution, the result is provided to the *Write* stage which (in general) writes a value to the regis-

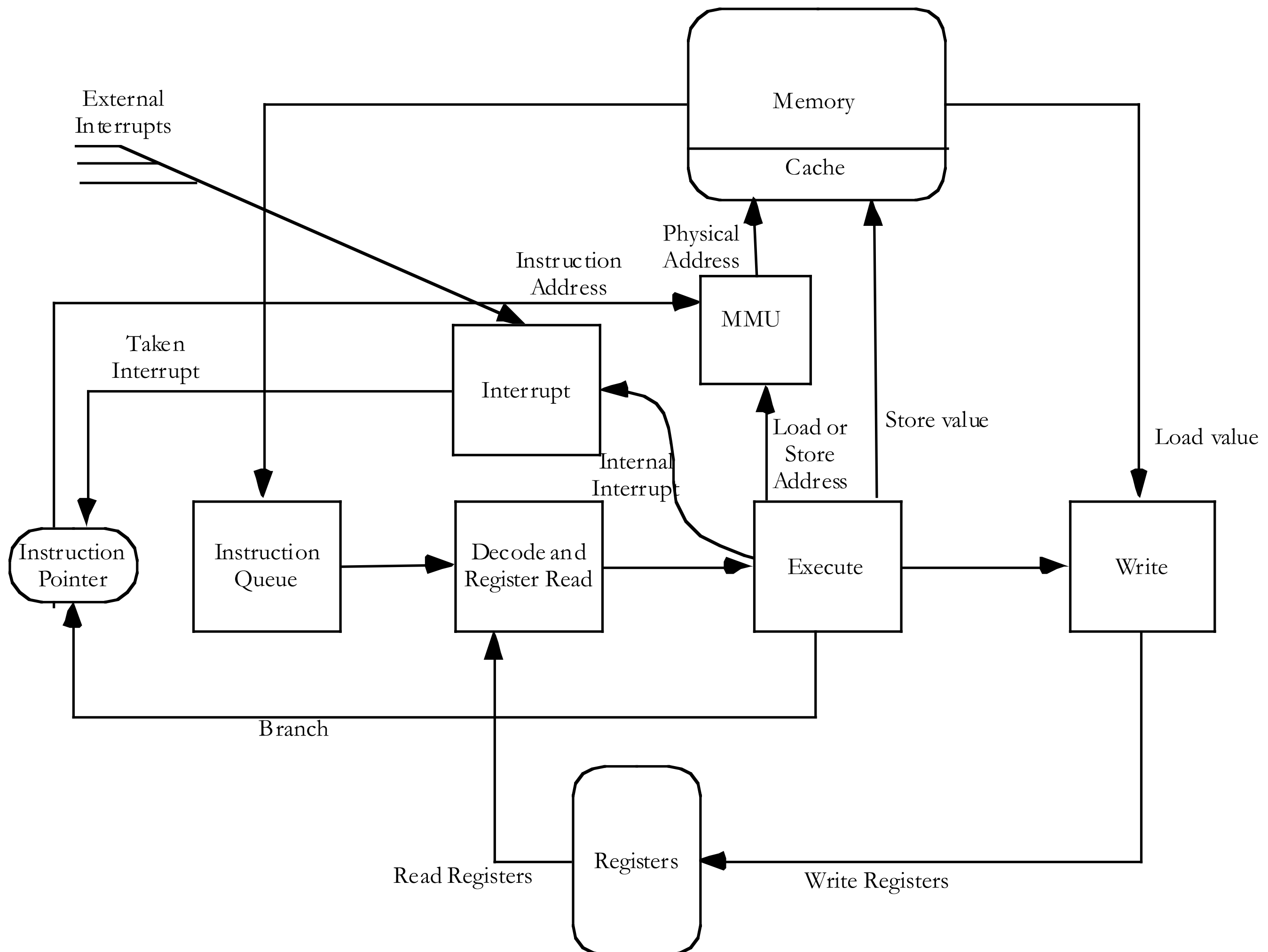

Figure 1 - K100 Pipeline Diagram

ter file. Not shown in the diagram is the forwarding of results done inside the Execute stage; in addition to writing the result to the register file, Execute also keeps a copy of its last result local to itself; this allows back-to-back dependent instructions to issue and execute at a rate of one per clock; in such circumstances, the source value for the dependent operand is obtained from the Execute forwarding buffer, rather than the register file. It is the job of Decode and Register Read to recognise the situation and set up control information appropriately.

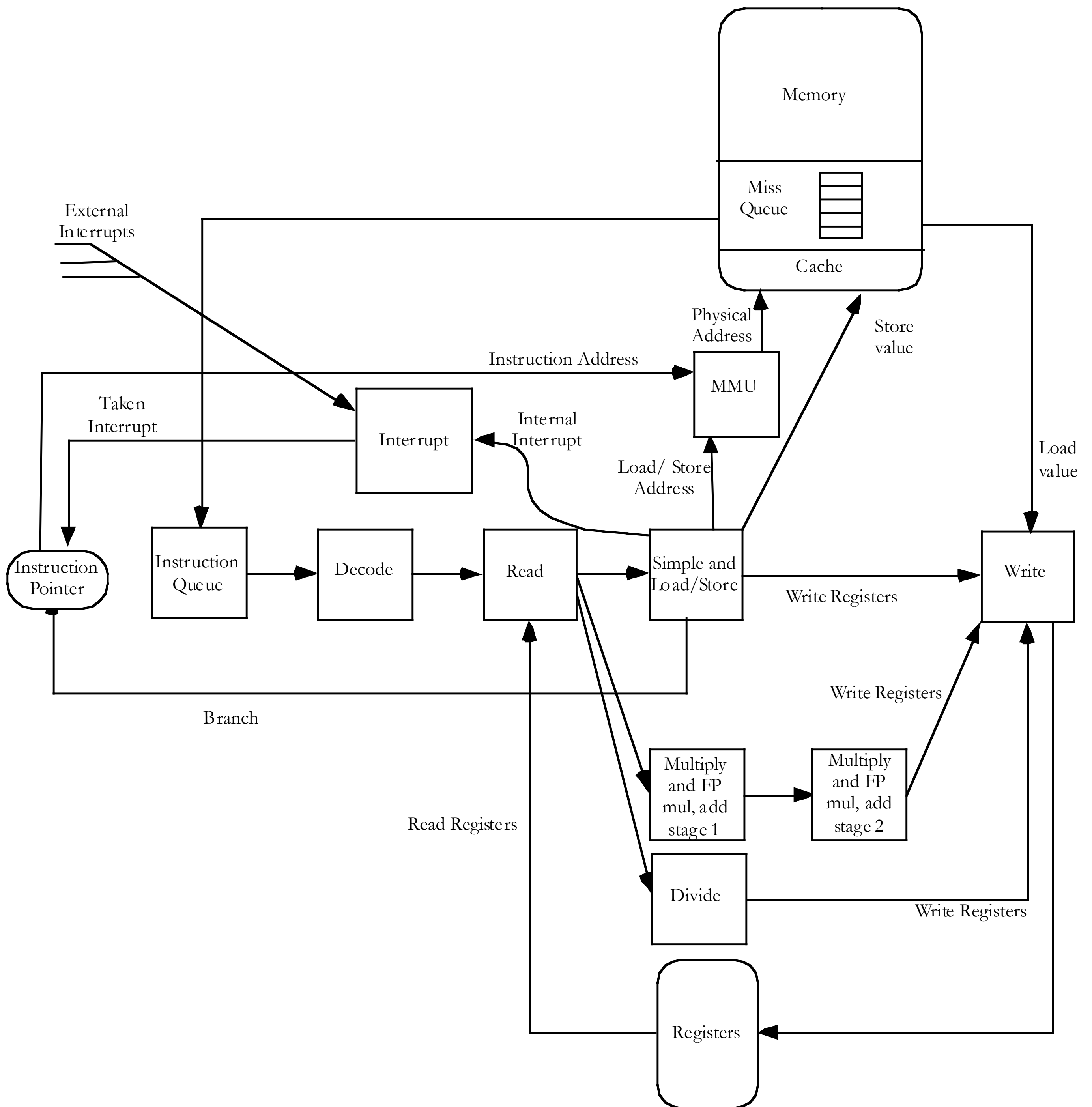

Figure 2 - K200 Pipeline Diagram

In the path from processor to memory, the diagram shows an *MMU*, and a cache as part of the memory system. The MMU's job is to translate virtual addresses to physical addresses; the *Cache* keeps copies of memory-resident data to reduce up the average access time to data.

Other paths are shown - branches cause a new value to be sent to the Instruction Pointer, resulting in the instructions queued in the Instruction Queue being discarded and new ones fetched (along with any necessary discarding of instructions in the pipe). And when an interrupt is to be taken, the Interrupt box sends a signal to the Instruction Pointer to select a new instruction address.

The Interrupt Box contains a small table of *Interrupt Vectors*, one entry for each interrupt source it can distinguish, which specify the execution address for the interrupt it is notifying to the processor. The Interrupt Box looks at all the interrupt requests outstanding, and signals the highest priority one to the processor.

K200

The K200 is still an in-order machine in the sense that it despatches or issues instructions in order; however, it can complete independent instructions out of order. It also has a longer pipeline than the K100, allowing a substantially higher clock rate. The K200 is no longer simply pipelined - it has a number of function units rather than just one execute stage; it separates out instruction decode from register read, and it uses forwarding only for simple operations. The multicycle operations (load, multiply, all floating point operations and divide) do not forward their results, but write them to the register file. To allow the machine to keep track of what's going on, each register is tagged with a busy bit - a register which is the destination of a multicycle operation is tagged as busy, and no instruction using a busy register can proceed past the register read stage. The busy bit is set when read and cleared when written[1], adding two clock cycles to the latency of these operations. The machine can also have multiple load operations running concurrently; not only are successive back-to-back loads pipelined through the cache, multiple cache misses are also allowed to be outstanding. The machinery keeps track of up to four cache misses in queue structures which hold destination register information, allowing returning values to be delivered correctly. Figure 2 provides a diagram of the K200.

K300

The K300 is an out of order machine, extending the internal resources of the K200 to form a traditional OOO machine with reservation stations, register renaming and a completion buffer but with otherwise identical structure to the K200. The hope with such a machine is that performance is gained through higher ipc and larger resources compared to a K200-like machine.

For a full description of how the approaches selected for the K300 work, and how they can help, the reader is urged to consult Hennessy and Patterson[2].

In the K300, there are more physical registers than architected. On every instruction which has a destination register, the K300 selects a free physical register to map to that register, and updates a lookup table of 'architectural to physical' register mappings; simultaneously, the machine maps the architected source register numbers specified by the instruction to their physical numbers. If the physical registers thus identified have valid content, the registers are read and provided along with the instruction for issue to the appropriate execution unit. If a source is not available, a note is made of the required phsyical register and the instruction sent to the

---

[1] This design choice is made simply for ease of description and (later) modelling; it is not recommended practice.

[2] Computer Architecture: a Quantitative Approach; John L Hennessy and David A Patterson; http://www.amazon.com/gp/product/1558605967/104-0869645-8150316?v=glance&n=283155

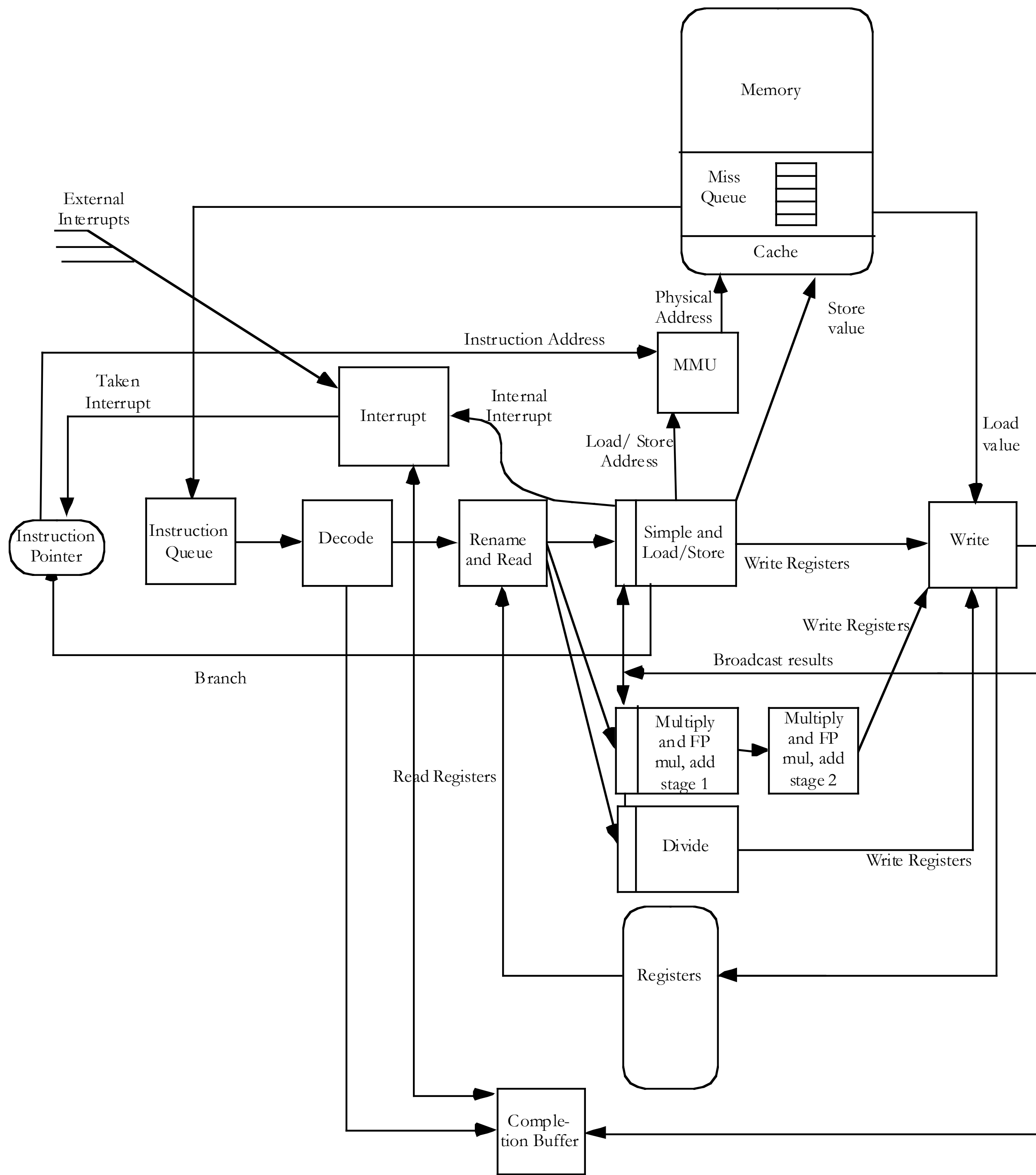

Figure 3 - K300 Pipeline Diagram

execution unit anyway. When a register is remapped, the old mapping is discarded and the corresponding physical register freed up. The physical registers are tagged with busy bits, as in the K200.

Each execution unit has a short buffer at its front end. Instructions are sent to these buffers (or reservation stations), where they remain until all the operands needed are available; as soon as all operands for a buffered instruction area available, it is provided to the execution unit proper for execution.

Decode also places each instruction, in program order, into the Completion Buffer. Instructions are retired from the Buffer in program order, as they successfully complete and write their results. An instruction which causes an exception has its Completion Buffer entry marked appropriately, and when the instruction is retired, the exception is taken. The CB simplifies handling of branch prediction; instructions down the predicted path are fetched and issued as usual; if it is discovered that a branch was incorrectly predicted, then it and any successors in the CB are discarded and the correct sequence of instructions fetched.

The use of register renaming and reservation stations allows the machine, in principle, to run at a higher level of efficiency. This comes about because - up to the limit of its internal buffers/queues - the machine can keep issuing instructions even when it cannot execute them yet. Performance improves because the machine can release instructions from a reservation station into the corresponding execution units concurrently - the machine could start a simple operation, a floating point operation and a divide simultaneously, for example. Were the machine not to have fetched those instructions and placed them in the reservation stations, it would have stalled on the first un-issuable instruction, and would have taken many more clocks to get the instructions issued. This class of microarchitectural improvement provides performance enhancements only when the latencies associated with instructions are fairly small, as is the case for simple and floating point operations and cache hits. A cache miss to slow DRAM can require a hundred clocks to satisfy the miss; this (even if the instruction stream supported such a degree of instruction-level concurrency) would require the completion buffer to hold a hundred entries, and for there to be a very large number of physical registers. All this adds area, power and complexity, all these being unlikely to be repaid in a performance increase in keeping with the area increase[3].

A 'real' K300 would probably also be a dual-issue machine; we keep the K300 single issue for the same reasons as aspects of the K200 - simplicity in description and modelling.

System Implementation

The system is a combination of processor, DMA and devices, stitched together with interconnect. In our example system we will use a simple split-transaction, out of order bus interconnect, and our example system will contain a processor, a DMA, a small number of I/O devices and a memory with integrated memory controller. A block diagram is given in Figure 4.

The system comprises a processor, a DMA engine, a memory controller and two I/O devices, connected up via a split-transaction bus. An agent can create several types of transaction - examples are a read from memory and a write to memory. For a read, the agent has to claim the address bus and when granted it, place an address on the bus. The data placed on the bus is a small 'packet' specifying the bus ID of the sender, the address of interest, the transaction required (here, a read, of one cacheline) and a transaction ID. The memory controller

[3] In fact the situation is worse than this, because to be able to keep the machine full of useful instructions, branch prediction must be very good. This also requires further expensive silicon and complexity which is (arguably) worthwhile for processors intended for computer applications; in most cases, in embedded, simplicity will pay off better.

(which, like all agents, continually monitors the address bus and the data in bus) will recognise that the packet is intended for it, accept the packet, free the bus, and (eventually) ask the memory itself for the required data. When the memory delivers the data, it will request the data in bus and when granted place packets on that bus - the number of packets required may be one or several, depending on the size of the memory read requested and the width of the data bus. Each response packet identifies the destination agent, the source agent, the transaction ID of the original request and the data payload. A write to memory by an agent works similarly, except that the initiating agent needs to claim both the address bus and the data out bus.

In addition to reads and writes, the interconnect can carry interrupts. An agent can interrupt another agent by sending it an interrupt message, which is formatted like a memory write (but with the transaction labelled an interrupt rather than a write). Agents may thus have several internal interrupt 'addresses' internally.

For clarity, the extra control lines needed to housekeep bus requesting and granting are omitted from the diagram.

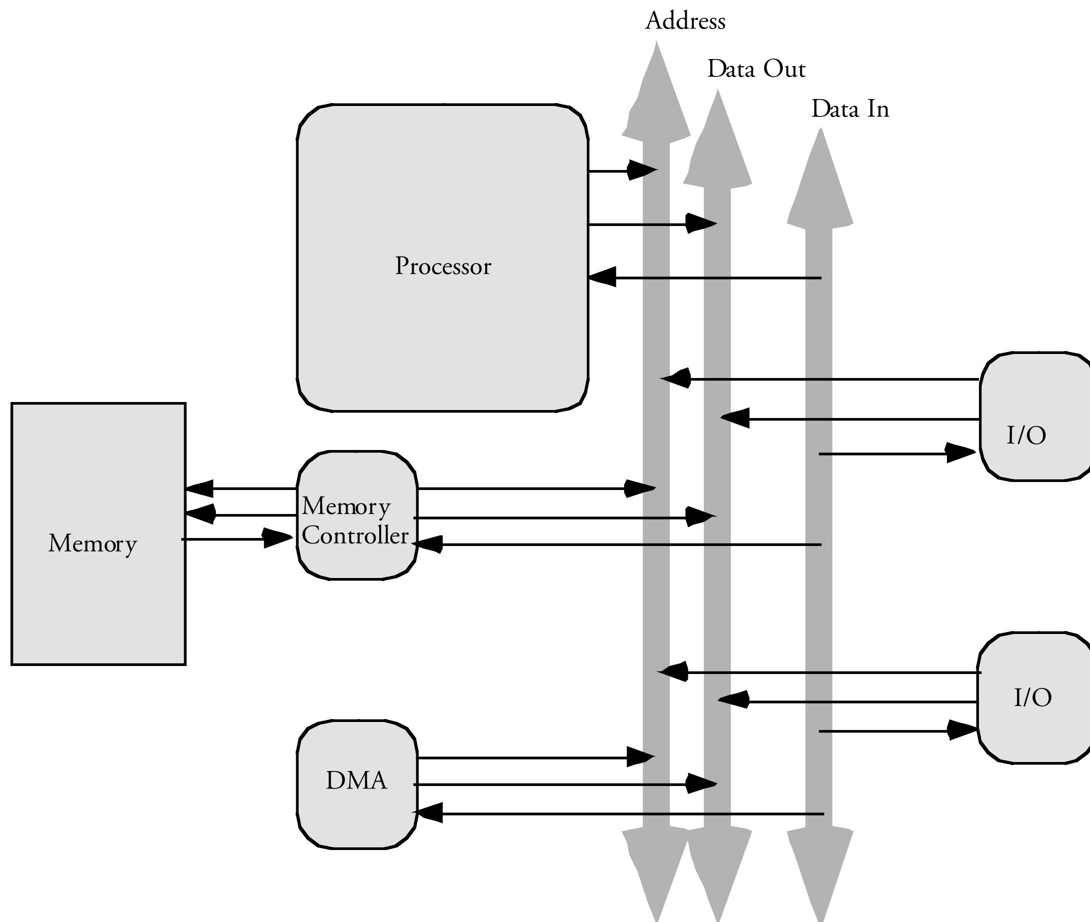

Figure 4 - Example System Diagram