Design and VLSI Implementation of an Access Processor for a Decoupled Architecture

Paul T. Hulina, Lizyamma Kurian,
Eugine B. John, and Lee D. Coraor

Department of Electrical and Computer Engineering
The Pennsylvania State University
University Park, PA. 16802
(814)-863-4253

pth@ecl.psu.edu

Abstract

Decoupled computer architectures provide high scalar performance by exploiting the fine–grained parallelism existing between the access and execute functions in a computer program. These architectures employ an access processor to perform data fetch ahead of demand by the execute process. Some of the decoupled architectures employ identical access and execute processors, but special processors to efficiently access data structures have also been proposed. In this paper, we present the design of an access processor targeted for VLSI implementation. The hardware elements, instruction set and instruction formats of the processor are presented in sufficient detail. The various trade-offs involved in the design are explained. Details of implementation of a scaled down version of the processor are presented.

* *This work was supported in part by the National Science Foundation under grant number MIP–8912455.*

1. Introduction

Decoupled computer architectures exploit fine–grain parallelism existing between access and execute functions in a computer program. The memory access task and execution task required for accomplishing a job are actually two interrelated but independent processes. Decoupled computer implementations attempt to separate these two processes to provide concurrency between them and to give high performance and increased flexibility.

Some separation of access and execute functions has been a feature of several early high performance computers such as the IBM 360/370, CDC 6600, CDC 7600 and CRAY–1. Later, several computer architectures have been proposed and some built, which partition access operations and computation functions in a program more distinctly, PIPE [5], SMA [7] and ZS–1 [11] [12] being typical examples. The Parallel Instruction Pipelined Execution (PIPE) machine is a decoupled architecture developed at the University of Wisconsin, Madison. It contains two identical VLSI processors; one serves as an access processor while the other serves as the computation processor. The Structured Memory Access (SMA) architecture from the University of Illinois, Urbana Champaign, consists of a Memory Access Processor (MAP) and a Computation Processor (CP). A single instruction stream enters the MAP, which splits the stream, forwarding the computation instructions to the CP. All memory requests are generated and controlled by the MAP. One decoupled architecture that has been implemented is the ZS–1 by the Astronautics Corporation of America. The ZS–1 has two functional units, the A–unit and the X–unit. The A–unit is used for all fixed point operations and all memory address computations. The X–unit is used for floating point operations. There are architectural queues between the two units as well as between the processors and the memory. MAP-200 [2], DAE [9], SDP [8] and FOM [1] are other examples of decoupled architectures.

All these architectures consist of two processors, one generating a stream of addresses for read and write requests and the other operating on the data to produce results. The first processor is often referred to as the access processor and the second one as the execute or computation processor. The two processors communicate through architectural queues, which tend to maximize their independence and allow them to *slip* with respect to each other. *Slip* indicates the distance the access process is running ahead of the execute process. *Slip* is an architectural attribute that allows concurrency between the two processors, by allowing the access processor to forward information to the execute processor, while the execute processor performs operations on previous information. Decoupled computers have two instruction streams for the two processors. In some of the architectures, like the SMA and the ZS–1, the splitting of a single instruction stream into two streams happens at execute time, whereas in others like the MAP–200 and PIPE, the splitting occurs at compile time.

The access instruction stream typically moves ahead of the execution stream and when a program is looping, it often precedes the execution stream by at least one loop iteration. The architectural queues buffer the prefetched data. Since the access instructions can fetch data in advance of need, decoupled architectures are less sensitive to memory access delays. The length of the queues is a critical factor in a decoupled architecture since often the distance the access process can run ahead is limited only by the queues. A slow memory access path can thus be compensated for by using longer queues. This characteristic is likely to be important in VLSI systems where communication delay, especially with memory, is a critical factor in determining system performance. Systems with queues utilize given memory bandwidth more efficiently.

1.1 Overview

We describe a decoupled architecture in Section 2, and explain the interfacing of the two processors, working of queues, and other communication details. Section 3 gives the options and trade–offs in the design of AP. The elements of the AP, the instruction set and the instruction formats of the AP are described in this section. Section 4 describes the VLSI design of each element in the AP. In section 5, we describe the VLSI implementation details, discuss the various VLSI aspects, and explain the tools we used. We conclude the paper in section 6.

2. The Architecture

A decoupled architecture using two processors, an Execute Processor (EP) and an Access Processor (AP) is described here. As in standard DAE (decoupled access execute) architectures [9], the EP executes program instructions and performs all the data computations required by the program. All accesses of the data memory are done by the AP. To avoid memory contentions between the EP and the AP, each processor is equipped with a separate memory unit, as shown in Fig. 1. Computation instructions are fetched from the Instruction Memory Unit by the EP. Data operands needed in the EP computations are fetched from the data memory unit by the AP and furnished to the EP. Results of EP computations are transferred to the data memory by the AP. The AP instructions are stored in a memory separate from the global data memory unit but in the event of AP pin limitations, the AP instructions could come from the global memory.

The code is split at compile time in such a way that all computation instructions are put in the EP section and the address calculation and access instructions are stored in the AP section. At execute time two different instruction streams independently enter the two processors. When a process is initiated, both the processors start separately. As explained later, the AP and EP use tokens to ensure correct operation wherever synchronization is required.

The performance improvement that can be achieved by using a decoupled system can be clearly shown with an example. Given below is the source code of FORTRAN Lawrence Livermore Loop 3.

```
      q = 0.0
      do 3 k = 1,1000
3      q = q + z(k)*x(k)
```

The assembly code that would be generated for a typical uniprocessor for the above program is given below. For sake of simplicity, data size is assumed to be a byte here.

-

```
L3:   R1 ← 1000                ;loop bound
      R2 ← 1                   ;loop counter
      R3 ← 0                   ;initialize q
      R4 ← base of z(R2)       ;load z(k)
      R5 ← base of x(R2)       ;load x(k)
      R6 ← R4 * R5             ;z(k)*x(k)
      R3 ← R3 + R6             ;q = q + z(k)*x(k)
      R2 ← R2 + 1              ;increment loop index
      BLE R2,R1,L3
```

The corresponding code for a decoupled architecture may look like the following.

Access Processor	**Execute Processor**
R1 ← 1000	
R2 ← 1	R3 ← 0
L3a: R4 ← Base of z(R2)	L3b: R6 ← R4*R5
R5 ← Base of x(R2)	R3 ← R3 + R6
R2 ← R2 + 1	BNEOD L3b
BLE R2,R1,L3a	
PUT EOD	

The AP and the EP see entirely different instruction streams. The instruction stream of the EP consists of only the computation instructions. The AP performs address calculations, initiates data memory accesses, and furnishes the EP with the data it requires for the computations. The EP is free to perform its data computations while the AP waits for access requests to be satisfied by the memory. The AP should be running ahead of the EP in order to achieve any performance improvement since the data fetched by the AP is required for computations in the EP. If the EP has to wait for the data, no performance improvement will be achieved. If the AP can run at least one loop ahead of the EP, the EP never has to wait for data operands and thereby improving the system performance. In such cases, decoupled architectures eliminate the requirement for a data cache.

In the above problem, there is no storage of results in each iteration of the loop. If data storage is required in each iteration of the loop, other optimization techniques such as software pipelining [6] could also be used. In such a case, the data being stored could belong to a different iteration than the ones being fetched from the memory in that particular loop.

One might note that two new instructions, PUT EOD (Put End-of-Data) and BNEOD (Branch if no End-of-Data) appear here. They are used to coordinate the two processors. When accessing multiple-element data structures, the AP uses *End-of-Data* (EOD) tokens to separate batches of data (such as an array, a row of a matrix) and PUT EOD instruction can be used for the purpose. The BNEOD instruction is the counterpart of this instruction in the sense that whenever one processor puts an EOD token, the other processor could check for the token using this instruction.

The AP can deposit the token easily in problems with static data bounds. In problems with dynamic bounds such as the C library string copy (*strcpy*) and string compare (*strcmp*) where the end of the string is not known until it is actually encountered, the approach is different. To exploit the advantages of a decoupled architecture in such an application, the AP can fetch the string elements one by one without checking for the delimiter and the EP can perform the comparison. The EP will get indication about the end of string, while the AP would have fetched beyond the end of the string. In such a case, the EP can send an EOD token to the AP to stop its fetch operation and also to flush any unnecessary data it has fetched. Meanwhile, the EP can continue operations of its own, but it should not read any data until AP has flushed its queue. This can be accomplished if the EP waits for a token from the AP indicating that it has flushed the queue.

3. Design of an Access Processor

Most current computer applications are developed using structured high level programming languages such as Pascal, Ada etc. In such languages, information is generally handled in a structured form, e.g. as procedures and data structures. Since these structures are rarely supported by current computer architectures, compilers must generate a considerable amount of code to navigate through the data structures and to simulate the various data types and constructs (e.g. bound checking, type conversion etc) supported by the high level programming language. Such code must be inserted in the object code, which causes a considerable overhead on system performance. Since efficient address generation for data structures is a primary goal in obtaining high performance in a computer system, we designed an application specific AP to do the accessing instead of using a conventional processor. The architecture of AP and its instruction set is tailored to perform more efficient address generation.

3.1 Design Options and TradeOffs

The main issues under consideration while designing the AP are the following:-

1. Instruction Set
2. Instruction Formats
3. Addressing Modes
4. Register Set
5. RISC or CISC architecture

The primary concern is to design a simple instruction set which meets the needs of efficient address calculation and accessing. We decided to use a Load Store architecture. A large register set is incorporated to reduce the number of data memory accesses required. The large register set would assist in fast address calculation. The processor should include base registers, index registers and bound registers. These could be designed as three separate register sets or as a single set. For reasons of simplicity, we chose to have a single register set, each register of which could be used as a base register, index register or bound register. We decided to include an indirect indexed addressing mode since it helps to step through arrays and other data structures efficiently. All instructions are 32 bits in length. The source and destination addresses appear in corresponding fields in the instruction format. All these factors contribute to ease of decoding. The overall emphasis is on keeping the processor simple, so that faster, low density implementations are feasible. For instance, the access processor could be fabricated with Gallium Arsenide technology, which is fast but circuit densities achieved so far are low.

The interface between EP and AP is another major area of concern. As mentioned earlier, the communication is through queues. The two instruction streams run at their own speed, with the queues absorbing excess data. One option is to have 2 queues, a Read Queue and a Write Queue. The EP reads data from the Read Queue and stores its computation results into the Write Queue. Even the control information such as *End-of-Data* can be passed through the same queues. Another option would be to have an additional control queue for communication of control signals. In spite of using a third queue, we found that explicit instructions to check for the control tokens were required, which led us to conclude a two-queue implementation is as efficient as a three-queue system. The *End-of-Data* could be a token or a hard signal. We decided to have tokens, though it would necessitate using one more bit for each element of the queue. The decision to use tokens instead of a hard INTERRUPT and INTERRUPT ACKNOWLEDGE signal was made inorder to to allow maximum *slip* between the two processors.

3.2 Elements of the Access Processor

The main blocks constituting the AP are illustrated in Figure 2. The functions of these elements are described below.

1. *AP Instruction Cache*: The AP code exhibits significant locality properties due to loops and an instruction cache can capture a considerable part of the locality.
2. *Register Set*: The AP consists of thirty two 32-bit registers, all of which could be used as base registers, index registers or bound registers. Register R_0 always gives a 0 on a read and ignores writes.
3. *Read Queue, RQ*: RQ is a FIFO buffer used to hold data that are fetched from the data memory by the AP and needed for EP computations. This queue is also used by AP to communicate *End-of-Data* information to the EP, in cases where the AP knows the bounds of the data structure being fetched.
4. *Write Queue, WQ*: WQ is a FIFO buffer used to hold results that are generated by the EP and need to be stored in the data memory by the AP. In problems with data dependent (dynamic) bounds, the AP will not know the bounds ahead and they will be clear only after the EP checks for the delimiter. The EP can then send an EOD token to the AP via this queue.
5. *Arithmetic Unit, AU*: The Arithmetic Unit of the AP consists of an adder, a subtractor and a multiplier used to perform address calculations.
6. *Program Counter, PC*: The AP consists of a 32-bit program counter.

7. *Memory Address Register, MAR*: The Access Processor is comprised of a 32-bit MAR.
8. *Instruction Register, IR*: The Instruction Register of the AP is 32 bits in length.

The Read and Write Queues have a width equal to the word-length of the system, which is 32 bits in our case. In addition, each word of the queue contains an additional bit, called the *End-of-Data* token bit. The *End-of-Data* token bit of the head of the queue is treated as the EOD flag also. A read from the queues will not occur if the EOD bit at the head of the queue is set. The queue will be left unaffected in such cases. The BNEOD (Branch if no End-of-Data) instruction checks the EOD flag (or the token bit of the head of the queue) and if it is set, it clears the flag as well as the head of the queue.

3.3 The Instruction Set

The instruction set of the AP is designed to facilitate fast access to data structures. The instructions are classified into the arithmetic, data transfer, control, and queue manipulation groups. A three address format is chosen for the arithmetic instructions. The two source addresses as well as the destination address are explicitly specified in the instruction. The queue is treated as a register assuming that queue accesses can be completed in a single cycle. The bit pattern corresponding to the 32nd register is used to address the queue. The same bit pattern could be used for both the RQ and the WQ because one queue is read only and the other is write only for the AP. Any invalid operation on a queue (reading from an empty queue, writing to a full queue etc.) puts the instruction that caused it on hold.

(a) Arithmetic Instructions

1. ADDI Rd,Rs,Im.const (Add Immediate)

$$Rd \leftarrow Rs + Im.const_{15..0}$$

   The 16-bit constant is sign extended and added to the contents of general purpose register Rs and the 32-bit result is placed into register Rd.

2. ADDR Rd,Rs,Rt (Register Add)

$$Rd \leftarrow Rs + Rt$$

   The contents of register Rs and register Rt are added to form a 32-bit result. The result is placed into register Rd.

3. SUBR Rd,Rs,Rt (Register Subtract)

$$Rd \leftarrow Rs - Rt$$

   The contents of register Rt is subtracted from the contents of register Rs to form a 32-bit result. The result is placed into general register Rd.

4. MULR Rd,Rs,Rt (Register Multiply)

$$Rd \leftarrow lowerword(Rs * Rt)$$

The contents of registers Rs and Rt are multiplied. The lower 32 bits of the product are placed into register Rd. Since the obtained product is an address, and can be represented in 32 bits, the upper 32 bits of the product should be all zeroes. The instruction gives rise to an exception otherwise.

5. MULI Rd,Rs,Im.const (Multiply Immediate)

$$Rd \leftarrow lowerword(Rs * Im.const_{15..0})$$

The contents of register Rs is multiplied with the 16-bit immediate constant. The lower 32 bits of the product are placed into register Rd. Since the obtained product is an address, and can be represented in 32 bits, the upper 32 bits of the product should be all zeroes. The instruction gives rise to an exception otherwise.

(b) Data Transfer Instructions

6. LDX Rd,Rs,Rt (Load Indexed)

$$Rd \leftarrow [Rs + Rt]$$

Rs is used as a base register and Rt as an index register here. The contents of Rt are added to the contents of register Rs to form a 32-bit unsigned effective address. The contents of the location specified by the effective address is loaded into register Rd.

7. LDW Rd,Rs,Im.const (Load)

$$Rd \leftarrow [Rs + Im.const]$$

Rs is used as a base register here. The immediate constant is sign extended and added to the contents of register Rs to form a 32-bit unsigned effective address. The contents of the location specified by the effective address is loaded into register Rd.

8. LDIUP Rd,Im.const (Load Immediate to Upper)

$$Rd_{31..16} \leftarrow Im.const$$

$$Rd_{15..0} \leftarrow Rd_{15..0}$$

The 16-bit immediate constant is loaded into the upper two bytes of the register Rd. The lower two bytes are left unchanged.

9. SBX Rd,Rs,Rt (Store Byte Indexed)

$$[Rs + Rt] \leftarrow Rd_{7..0}$$

Rs is used as a base register and Rt as an index register. The contents of Rt are added to the contents of register Rs to form a 32-bit unsigned effective address. The least significant byte of register Rd is stored at the effective address.

10. SHX Rd,Rs,Rt (Store Half Word Indexed)

$$[Rs + Rt] \leftarrow Rd_{15..0}$$

Rs is used as a base register and Rt as an index register here. The contents of Rt are added to the contents of register Rs to form a 32-bit unsigned effective address. The least significant half word of register Rd is stored at the effective address.

11. SWX Rd,Rs,Rt (Store Word Indexed)

$$[Rs + Rt] \leftarrow Rd$$

Rs is used as a base register and Rt as an index register here. The contents of Rt are added to the contents of register Rs to form a 32-bit unsigned effective address. The content of register Rd is stored at the effective address.

12. SB Rd,Rs,Im.const (Store Byte)

$$[Rs + Im.const] \leftarrow Rd$$

Rs is used as a base register here. The immediate constant is sign extended and added to the contents of register Rs to form a 32-bit unsigned effective address. The content of register Rd is stored at the effective address.

13. SH Rd,Rs,Im.const (Store Half Word)

$$[Rs + Im.const] \leftarrow Rd$$

Rs is used as a base register here. The immediate constant is sign extended and added to the contents of register Rs to form a 32-bit unsigned effective address. The content of register Rd is stored at the effective address.

14. SW Rd,Rs,Im.const (Store Word)

$$[Rs + Im.const] \leftarrow Rd$$

Rs is used as a base register here. The immediate constant is sign extended and added to the contents of register Rs to form a 32-bit unsigned effective address. The content of register Rd is stored at the effective address.

(c) Control Instructions

15. JMP offset (Jump)

$$PC \leftarrow PC + offset$$

The 26-bit signed offset is added to the PC contents to obtain the branch address.

16. BEQ Rs,Rt,offset (Branch if equal)

$$condition \leftarrow (Rs = Rt)$$

if condition then

$$PC \leftarrow PC + offset$$

If the contents of registers Rs and Rt are equal, then the 16 bit signed offset is added to the PC to obtain the branch target.

17. BNE Rs,Rt,offset (Branch if not equal)

$$condition \leftarrow (Rs \neq Rt)$$

if condition then

$$PC \leftarrow PC + offset$$

If the contents of registers Rs and Rt are not equal, then the 16-bit signed offset is added to the PC to obtain the branch target.

18. BLT Rs,Rt,offset (Branch if less)

$$condition \leftarrow (Rs < Rt)$$

if condition then

$$PC \leftarrow PC + offset$$

If the content of register Rs is less than that of register Rt, then the 16-bit signed offset is added to the PC to obtain the branch target.

19. BLE Rs,Rt,offset (Branch if less or equal)

$$condition \leftarrow (Rs \leq Rt)$$

if condition then

$$PC \leftarrow PC + offset$$

If the content of register Rs is less than or equal to that of register Rt, then the 16-bit signed offset is added to the PC to obtain the branch target.

20. BGT Rs,Rt,offset (Branch if greater)

$$condition \leftarrow (Rs > Rt)$$

if condition then

$$PC \leftarrow PC + offset$$

If the content of register Rs is greater than that of register Rt, then the 16-bit signed offset is added to the PC to obtain the branch target.

21. BGE Rs,Rt,offset (Branch if greater or equal)

$$condition \leftarrow (Rs \geq Rt)$$

if condition then

$$PC \leftarrow PC + offset$$

If the content of register Rs is greater than or equal to that of register Rt, then the 16-bit signed offset is added to the PC to obtain the branch target.

(d) Queue manipulation instructions

22. PUT EOD (Put End-of-Data)

$$RQ \leftarrow EOD$$

The AP deposits an EOD token to the Read Queue. This marks the end of the data structure (or substructure) the AP was fetching.

23. BNEOD offset (Branch if no End-of-Data)
if EOD flag is not set,

$$PC \leftarrow PC + offset$$

The 26 bit signed offset is added to the PC contents to obtain the branch address if the EOD flag is not set.

24. FLUSH RQ (Flush Read Queue)
The Read Queue is flushed and the queue counter reset. If EP makes an attempt to read the queue now, the instruction cannot be completed.

3.4 Instruction formats

All instructions are 32 bits in length. A 5-bit opcode is required and a 5-bit combination is adequate to address any register or queue. The different instruction formats required are illustrated in Figure 3. In the immediate arithmetic operations, loads and stores with immediate offsets and conditional branches, two register addresses are specified and there is a 16-bit immediate constant. For register add, subtract, multiply, and the indexed load and store operations, 3 register addresses are specified in the instruction. The JMP and BNEOD instructions accomodate a 26-bit immediate constant. The FLUSH and PUT EOD instructions do not require any addresses or offsets.

The EP can have an instruction set compatible with the present RISC architectures. In addition, EP also should have queue manipulation instructions as in the AP instruction set. The AP instruction set could easily be a subset of the EP instruction set.

4. VLSI Design of AP Elements

There are several important aspects to be born in mind when designing a computer architecture for VLSI implementation. The cost of manufacture of an integrated circuit depends on chip area (A). More important is the fact that die defects make costs increase more rapidly than O(A) and chips above a certain area cannot be practically manufactured. Pin requirements of a chip is another important determinant of chip cost. Every wire that connects a pin on the package to a bond pad on the chip is a potential source of a defect and yield of packages with large numbers of pins is usually small. So chips with large numbers of pins are generally expensive. Power dissipated by a chip affects the design of the packaging of the chip. If there are a number of fast switching outputs, large amounts of heat could be generated; plastic packaging might not suffice and ceramic or other expensive packaging could be required.

VLSI designs should be regular structures since the time and effort required to produce a design depends on the number of different elementary cells required more than the total number of transistors used. For instance, it is quicker to produce a large register bank by replicating a properly designed single register bit than generating a random logic function which uses far fewer transistors. Random logic can also be implemented in regular structures such as the PLA or ROM, but an implementation based on gates has higher speed. The intermediate approach is to use a cell library that consists of standard gates, latches, flip-flops etc. The designer picks up appropriate cells and wires them together. This might not result in the smallest possible layout, but the speed can be good. Designing circuits with testability aspects in mind is also important since testing costs can be a significant portion of the total device costs.

The various elements of the AP are designed keeping the above factors in mind. The designs of the instruction cache, read and write queues, and register bank are described below.

4.1 Instruction Cache

On–chip cache memories provide a good path for alleviating the bottleneck produced by external communication. However, they must be small enough to be implementable on–chip, yet large enough to be functionally useful. Fully associative caches do perform well, but these caches require complex and expensive cache controllers which limit their use on–chip. There have been several attempts to reduce the circuit complexity and increase the hardware efficiency of cache memories in order to implement them on–chip. The queue-based instruction cache proposed in [3] is an area efficient cache architecture which makes it suitable for on–chip implementation. It consists of a RAM queue, a set of registers (for directory), and the associated control logic, as illustrated in Fig. 4. The data is loaded from main memory into the RAM queue in a FIFO fashion. However, the CPU may read any of the words currently in the queue and in any order. So the queue approach concerns only the way data is loaded into the cache. Unlike other cache designs, there is no concept of a line (block). The whole RAM is used as one block (or segment) and the address space of the resulting cache is contiguous. This architecture is similar to traditional instruction buffers in some respects.

The operation of the cache depends on three load procedures. A *LoadNew* procedure is used to load data into an initially empty cache or to clear an old cache and reload it with new data. To load data sequentially from main memory into a non-empty cache, a *LoadSequential* procedure is used. One major feature of the queue-cache is *preloading*. Preloading means that when the CPU requests information from an address that is not in the queue, but not too far from the last address loaded into the queue, then the referenced data as well as those in the intermediate addresses are fetched and appended to the queue without clearing the queue. Whenever a forward branch is encountered, the control logic checks to see how far ahead the new branch location is from the last address loaded in the queue. If this distance is within a predefined *preload window*, then the *cache preload procedure* is invoked. Preloading thus prevents unnecessary flushing of the queue, which occurs in the traditional instruction buffers.

Previous simulation studies [14] have shown that these queue-based caches are as effective as fully associative caches for small cache sizes, while they consume much less chip area. The queue–cache is thus very suitable for a VLSI implementation. Hence we decided to have a queue-cache for our AP.

4.2 Read and Write Queues

The read and write queues are constructed from shift registers with right shift capability as illustrated in Fig.5. An n bit queue of m elements can be constructed from n shift registers of size m. The right end of the registers or the least significant bit of all the shift registers together form the head of the queue. The head of the queue is thus fixed and the tail of the queue moves depending on the number of elements. There is an updown counter which gets incremented for each write and decremented for each read and thus it always indicates the current tail of the queue. There is a decoder or address selector which controls the write to the tail of the queue. A read occurs from the head and is automatically accompanied with a shift. There are two signals called the *queue empty* and *queue full* which prevent reading an empty queue and writing to a full queue. Whenever an instruction involving the queue is issued, the control unit of the queue checks for the queue empty or full signals. An N bit counter and an N to 2^N decoder together with n shift registers of size $2^N - 1$ would yield an n-bit queue of length $2^N - 1$ as illustrated in Fig. 5.

4.3 Basic Queue Cell

A flip-flop with a multiplexer at the input which can select the data bus or the output from the previous cell as the next input to be loaded would suffice as the basic queue cell. An array of such cells can be arranged to obtain the read and write queues of desired size and width.

4.4 Bus Structure

The data path was designed by inspection of the instruction set, with the aim of implementing all the functions required by each instruction in the smallest possible number of cycles with the simplest bus structure that would work. Single cycle execution of all instructions is targeted. The speed of load and store instructions would however depend on the memory speed. To achieve fast execution of the address calculation instructions, three internal buses are utilized, as shown in Fig. 6. Two registers can simultaneously be read to two buses, which form the two operands for the arithmetic.operations. For the immediate instructions, the immediate constant is directed to the AU from the IR through one of the buses and the other operand through the other bus. The AU output is connected to the output bus through tristate inverters. The buses that form the input to the AU are called *abus* and *bbus* and the output bus is called *obus*. Also shown in Fig. 6 is the external data bus DBUS and the address bus ADBUS.

4.5 Register Bank

The register bank consists of thirty two 32-bit registers. As mentioned before, the register bank uses three buses; two are used to read operands and the third is used to write the AU result into a register. Three decoders are required, one to select the destination register and two to choose the source registers for any operation. Decoders with enables are used so that the source and destination addresses could be permanently connected to the decoder and the control unit could enable or disable the decoder. The decoders activate the appropriate register according to fields in the current instruction. A 2-phase nonoverlapping clock could be used, with source registers being read in the first phase of the clock and the result being loaded into the destination register in the second phase. The floor plan of the register bank is shown in Fig. 7.

4.6 Basic Register Cell

The basic register cell consists of a storage element with two tristate outputs that can be connected to two buses. Each cell has one write enable and two read enables. The value at the data input gets written into the storage element if the write enable is activated and the stored value becomes available at the *abus* or *bbus* depending on which read enables are activated. Thirty two such cells are connected together to form a register and thirty two such registers form the register set.

4.7 The Arithmetic Unit, AU

The Arithmetic Unit of the AP consists of an adder, a multiplier and a subtractor (comparator). The two operands for each arithmetic operation are fed to the AU via the *abus* and *bbus*. The result is transferred to the register bank through the *obus*. The comparator outputs are written into flags which are tested for conditional branches.

4.8 AP Control Unit

The control unit can be implemented in microcode or be hardwired. The following are the control signals required for the chip.

1. Source register 1 enable
2. Source register 2 enable
3. Destination register enable
4. Instruction register to abus
5. DBUS to IR
6. WQ to DBUS
7. DBUS to RQ
8. Write to RQ
9. Read from WQ

10. Reset RQ
11. Increment PC
12. PC to obus
13. abus to PC
14. obus to MAR
15. MAR to ADBUS
16. Adder output to obus
17. Multiplier output to obus

4.9 Pin Requirements

The external connections that the AP needs are listed below. It may be noted that the two queues are multiplexed to the same port. This reduces the pin requirement and does not compromise any performance, since the EP reads from or writes into only one queue at a time. Though in Fig. 1, the AP is shown to have separate data and address buses to instruction and data memory, a common bus is assumed here due to pin limitations.

Pin(s)

1	Vdd (Power Supply)
2	GND (Ground)
3	ϕ 1 clock
4	ϕ 2 clock
5	Reset
6-37	$DBUS_{31-0}$ (Data Bus)
38-69	$ADBUS_{31-0}$ (Address bus)
70-101	Queue port

4.10 AP Floor Plan

The floor plan of the chip is shown in Fig. 7.

5. VLSI Implementation

We implemented a scaled down version of the access processor described above as a student project in a VLSI course. This version has 8-bit address and data lines and has eight 8-bit registers. The methodology is to implement the 8-bit version and then analyze the scalability of the design to find out how much more complex the 32-bit version would be. We can thus study the VLSI feasibility of our design.

5.1 Implementation Details

The instruction format was restructured to suit the 8-bit implementation. All instructions are of length 16 bits. A 5-bit opcode is required and a 3-bit combination is adequate to address any register. The 8-bit implementation thus requires a 16-bit Instruction Register. The PC and MAR in our experimental implementation are 8 bits long. The implemented version of the AP had exactly the same bus structure as detailed in the full scale design.

The basic register cell to suit the above described three bus structure was designed using gates in the standard cell library. The pads to connect to the buses were placed in such a manner that the interconnections while constructing the register bank would be regular and would consume least area. The register bank was constructed replicating the basic cell designed above. Wires do consume a lot of area in the register bank. All efforts were made to keep it to a minimum. There are three decoders also, one for each bus. These decoders are also implemented using gates from the standard cell library. They are then placed adjacent to the register set.

Both the Read and Write Queues of the implemented version are 8 bits in width and have a length of eight. They are constructed by replicating the basic queue cell designed using gates and flipflops. The designed queue is shown in Fig. 8. Most of the control logic is implemented using gates from a standard cell library. Since there is very little regularity to be exploited in the layout, this approach helps to generate the design in a reasonable amount of time. The standard cell based designs are easy to modify also.

The scaled down version of the chip needs 29 pin connections considering single data and address buses in precisely the same manner as designed for the 32-bit version. Fig. 9 illustrates the implemented chip.

5.2 VLSI Aspects

Regularity, modularity, scalability, chip area, wire area etc. are major aspects in a VLSI implementation. We exploited all regularity in building up the chip. We designed only the basic register cell and replicated the structure to construct the Register Set, PC, MAR and IR. Three decoders were required for the register set and they were also repetitive instances of one decoder that was originally designed. Similarly, we designed only a basic queue cell and then used replication to make the queue.

The design was done in highly modular fashion. This helped in developing the chip, module by module, doing sufficient testing at module level before integrating and removing bugs at sub–module level itself. Another factor that had to be taken care was the I/O requirement for each sub-module. We designed modules in such a way that area for pads for each module and the wire area for interconnection could be minimized.

Regarding area/time trade-offs, reducing time was the main emphasis. We traded area for speed wherever the choice had to be made. An example is the use of an area expensive combinational multiplier which would perform the multiplication fast. We wanted to achieve single cycle execution of all instructions.

Another aspect in a VLSI implementation is scalability. It is particularly important here, since the original design was for a 32-bit processor, but only an 8-bit version was implemented. From the 8-bit implementation, we can speculate about the complexity of the 32-bit AP. We found that the PC, MAR, IR, Register Set, and Queues of the 32-bit chip could be constructed with linear increase in complexity. The multiplier has $O(n^2)$ complexity. The area bound of the chip (except the combinational multiplier unit) could be O(n). Since our design of instruction formats and bus structure, alllows single cycle instruction execution, we can consider the chip as having O(1) time bound.

The important lessons that we learned from the implementation are the following: (1.) The design of each module has to be done with VLSI implementation in mind. (2.) Each sub-module should be controlled locally rather than globally, to the maximum extent as possible. This reduces the I/O requirements for each module and reduces the routing complexity considerably. So basic cells and smaller building blocks should be designed with I/O in mind.

5.3 VLSI Tools and implementation strategy

The chip was developed using *Octtools* [13]. Octtools supports a Pascal–like high level language called *bdsyn* to facilitate the design. One can specify the logic of the desired circuitry in *bdsyn* and the hardware can be automatically generated using the various tools. There are several minimization tools that can be used for generation of efficient circuits. The major disadvantage associated with *bdsyn* is that there is no control over the type of hardware generated. Also *bdsyn* can be used only for combinational logic. Since we wanted to specify the logic from gate level, we used another octtool facility, the *bdnet*, more often than bdsyn. Logic gates and flip-flops of the designer's choice can be interconnected with *bdnet* to generate precisely the exact hardware that is desired. Again, *bdnet* is the tool to integrate the sub–modules to yield larger modules and finally the large modules to the chip. The octtools simulation tool *musa* is used to test the circuits at the module and integrated levels. Octtools contain a variety of tools to efficiently route and place the circuits. For example, the Octtool suite contains *mosaico*, a macro cell router, *padplace*, a pad placement tool, *Timberwolfsc*, a package to do standard cell placement and routing, *mizer*, a minimization tool etc. These tools are used iteratively to minimize wire and routing area.

6. Conclusion

We designed an access processor which can be interfaced to a conventional processor in a decoupled configuration. We explained the overall architecture of the system and the design of the AP in specific. The instruction set, instruction formats and hardware elements of the AP are explained. The whole design is targeted for a VLSI implementation and the various trade-offs involved in the design process are explained. A scaled down version of the design was implemented in VLSI and we detail the implementation strategy, the tools used and the experiences during the process of implementation. The implementation demonstrated the feasibility of a processor with architectural queues.

We did not consider any pipelining in the implemented version, but pipelining of fetch-decode-execute stages can be done to improve the performance. The 32-bit access processor could be used along with a standard RISC processor (with added queue manipulation instructions) as the execute processor to form a usable decoupled architecture.

References

1. Brantley W.C., Weiss J., "Organization and architecture tradeoffs in FOM", presented at IEEE Intl. Workshop Comp. Syst. Organization, New Orleans, LA, March 1983.
2. Cohler E.U., Storer J.E., "Functionally parallel architectures for array processors", IEEE Computer, vol.14, pp.28-36, Sept.1981.
3. Coraor L.D., Hulina P.T. and D.N. Mannai, "A Queue-Based Instruction Cache Memory", Proc. of the International Symposium on Computer Architecture and Digital Signal Processing, October 1989, Hong Kong, pp.281-286.

4. Furber Stephen B., "VLSI RISC Architecture and Organization", Marcel Dekker Inc., New York and Basel, 1989.
5. Goodman et al, "PIPE: A VLSI Decoupled Architecture", 12th Annual International Symposium on Computer Architecture, June 17-19, 1985, Boston, Massachusetts, pp.20-27.
6. Hsieh J.T., Pleszkun A.R. and Goodman J.R., "Performance Evaluation of the PIPE Computer Architecture", Technical Report #566, Computer Sciences Department, University of Wisconsin, Madison, Nov. 1984.
7. Pleszkun A.R, Davidson E.S., "Structured Memory Access Architecture", Proc. Intl. Conf. on Parallel Processing 1983, pp.461–471.
8. Shivley R.R., "Architecture of a Programmable Digital Signal Processor", IEEE Transactions on Computers, vol.C–31, Jan. 1982.
9. Smith J.E., "Decoupled Access/Execute Computer Architecture", ACM Transactions on Computer Systems, Vol.2, No.4, November 1984, pp 289-308.
10. Smith J.E., Weiss S. and Pang N.Y, "A Simulation Study of Decoupled Architecture Computers", IEEE Transactions on Computers, Vol.C-35, No.8, August 1986.
11. Smith W.M, Abraham S.G. and Davidson E.S, "A performance comparison of the IBM RS/6000 and the Astronautics ZS-1", IEEE Computer, January 1991.
12. Smith J.E., "Dynamic Instruction Scheduling and the Astronautics ZS–1", IEEE Computer, July 1989.
13. Octtools Users' Guide, University of California, Berkeley, 1990.
14. Mannai D.N., Hulina P.T., Coraor L.D., Kurian L., "Design and Performance Evaluation of a Queue-Based Instruction cache Memory", Computer Engineering Technical Report TR-90-081, The Pennsylvania State University.