# DRAFT: A DYNAMICALLY RECONFIGURABLE PROCESSOR FOR INTEGER ARITHMETIC

Donald M. Chiarulli, W. G. Rudd, and Duncan A. Buell[1]
Department of Computer Science
Louisiana State University
Baton Rouge, Louisiana 70803

## ABSTRACT

A special computer for high-precision arithmetic features an ALU that is dynamically reconfigurable under program control. The 256-bit ALU consists of 8 32-bit slices each of which has its own ALU operation code in each microinstruction. The slices can remain logically separated from each other, or be dynamically connected to either or both of their neighbors under control of a segment control code that is part of each microinstruction. The micro-assembly language designed for the machine includes special features to assist in the control of the segmentation, data addressing, and control sequencing. Estimations of the times required to execute arithmetic operations on the machine show that it will be exceptionally fast for problems in computational number theory and factoring of integers.

## INTRODUCTION

> ". . . The dignity of the science itself [factoring numbers] seems to require that every possible means be explored for the solution of a problem so elegant and so celebrated[1]."

The DRAFT (Dynamically Reconfigurable Architecture for Factoring Things) computer now under construction at LSU represents the first in a new family of computers that feature dynamically reconfigurable CPUs [2]. Under program control these machines can alter their effective wordlengths to suit the sizes of the current operands. Each slice of the ALU has its own operation code so that the machine can execute several different instruction sequences simultaneously on different sets of operands of different lengths. The result is a highly parallel machine. But all the ALUs are under control of a single sequence controller with a single clock. Furthermore, all ALUs use the same data store address in executing a micro-instruction. The result is that the machine does not suffer from the problems in controlling and coordinating asynchronous processors that arise in true multiprocessor architectures.

The original motivation for building the IRAFT machine is to support Buell's research [3,4,5,6] in computational and experimental number theory. Problems in this area of mathematical research include primality testing, the factoring of large numbers, encryption and decryption, and the study of the structure of the set of integers. The unifying characteristic of such problems is that they require fast execution of integer arithmetic on large (75 decimal digits is not unusual) integer operands.

Researchers at the University of Georgia have adopted another approach in this area [7,8,9]. They have built and are now using a machine designed solely for rapid execution of the Brillhart-Morrison continued fraction algorithm for factoring large numbers [10]. The machine, called EPOC, is basically an array processor for integer arithmetic.

Our design philosophy is that, in all stages of design and construction of the machine, we will place equal weight on considerations of hardware, software and algorithms. This approach is absolutely essential if we are to develop a system that will satisfy our objectives.

In keeping with this philosophy, we present in this paper a discussion of all three of these areas as they relate to the DRAFT machine. We begin with an overview of the hardware design. In part 2 we introduce a micro-assembler that uses a pseudo-register technique to simplify microprogramming the machine. We include two programming examples. In part 3 we derive some performance estimates for the prototype machine. Part 4 offers a summary and some our plans for further research.

## 1.0 The DRAFT architecture

The heart of the DRAFT machine (Fig. 1) is its 256-bit ALU, which is constructed from eight 32-bit ALU slices. At the micro-instruction level, each slice can be combined with or isolated from its neighbor on either side. An isolated slice or a connected set of neighboring slices is called a segment. There is no restriction on how the slices can be combined to form segments. A single 256-bit segment, eight 32-bit segments, 2 32s, a 64 and a 128, or any other intermediate combination can be formed by appropriate setting of the segmentation control word that is a part of each micro-instruction.

Fig. 2 is a sketch of the DRAFT micro-instruction format. Each micro-instruction includes a sequencer control section, a segmentation control word, and an ALU control section that contains one ALU control word for each slice. Each ALU slice has its own operation code, operands, and set of condition test masks. But the ALU slices are not independent concurrent processors. A single

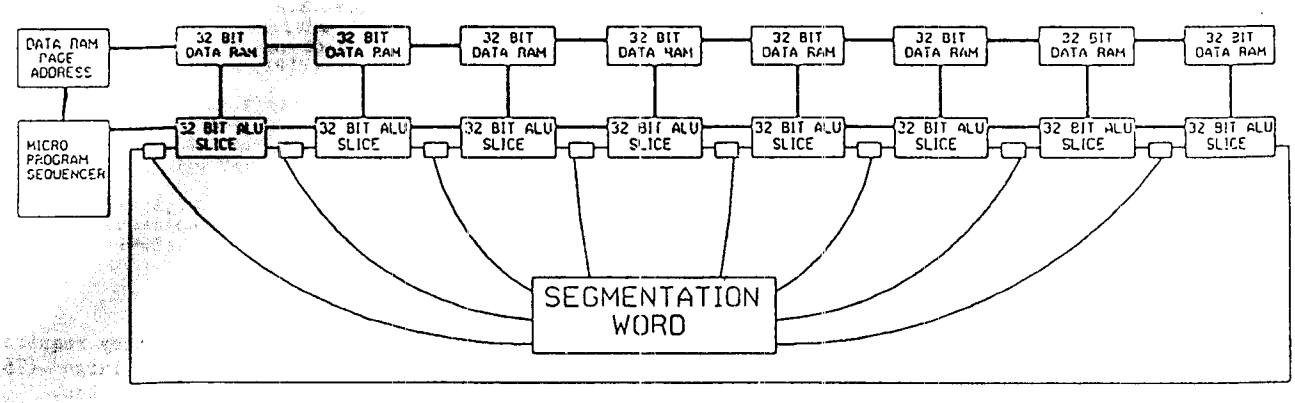


FIGURE 1 - Draft Machine Segmentation

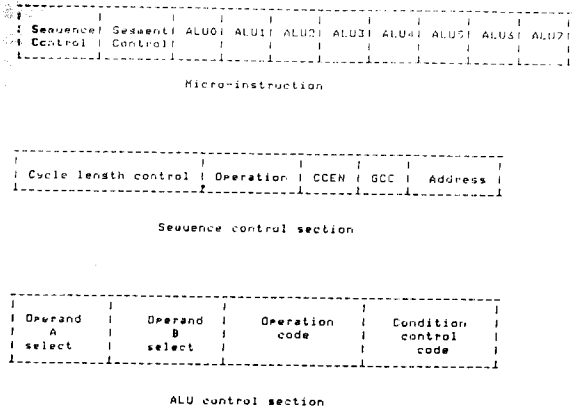


Figure 2. DRAFT micro-instruction format

sequencer and clock control sequencing and timing of all operations.

The control and data stores in the DRAFT machine are separate. The first version will have a 4K control store for 252-bit micro-instructions. The data store will consist of 4K pages, each of which contains 16 256-bit words. The data store page is selected by a 12-bit field in the sequencer control section of the micro-instruction. This field is also used for the control store address in those micro-instructions that refer to the control store.

The architecture offers a modified multiple data stream and many of the capabilities of a multiple instruction stream machine, but the design eliminates many of the concurrency problems found in conventional MIMD machines. The dynamic segmentation feature makes it especially appropriate for the kind of problems for which it is designed.

The entire machine will be constructed from "off-the-shelf" LSI devices. We estimate that the total cost for hardware will be about $25,000.

The DRAFT machine will be attached to Q-bus ports on a PDP 11/23, which will serve as the front end and console for the DRAFT machine. All I/O for the DRAFT machine will be through direct transfers through the Q-bus to and from the DRAFT machine's control and data stores. We will use separate Q-bus ports for control and status information. The PDP 11 is connected to our VAX 11/780, on which the software development work is being done.

We now describe the DRAFT architecture in more detail.

1.1 The micro-instruction sequencer

The micro-program sequencer is designed around the AMD 2910A Micro-program Controller. This controller chip provides 16 instructions for condition testing and branching, a counter register for loop control, and a 9-level stack for subroutine linkage. The DRAFT sequencer operation code is a 6-bit field formed from the 4 bit 2910A instruction input (SIO-SI3), a condition enable bit (CCEN), and a global condition combination (GCC) bit. The CCEN bit distinguishes between the unconditional (CCEN = 0) and conditional (CCEN = 1) analogs of the 16 2910A operations. The GCC bit defines the logical operation (AND/OR) used to combine the local condition codes from each segment to generate the condition code input to the 2910A. A GCC bit set to one (AND) means that all the segments must return a logic 1 from their condition code outputs in order for the global condition to be true. A GCC value of zero yields a global condition true if any of the segments produces a local condition that is true. In combination with the CCEN input this system generates three analogs to each 2910A operation that tests the external condition input to the 2910A.

For example the JSB (jump to subroutine) sequencer operation has the analogs

```
GCC=X  CCEN=0   JSB    : unconditional JSB
GCC=1  CCEN=1   JSBAND : conditional JSB and
GCC=0  CCEN=1   JSBOR  : conditional JSB or
```

In the sequencer mnemonics listed in Table 1, the mnemonics for the conditional operations are formed by adding AND or OR to the end of the corresponding mnemonics for the unconditional operations.

The data page address is a shared field used both by the 2910 sequencer as a direct address operand for all instructions except the EX instruction, and by the ALU slices as input to the data RAM workspace register during an EX instruction. The data RAM workspace pointer can be changed only by sequencer instructions that do not cause a transfer of control. This is a programming inconvenience but should not be a major restriction on the architecture.

In traditional sequencers, a fixed-frequency clock signal generates the master clock frequency and hence the micro-cycle length. The propagation delay for the operation with the longest path for data flow through the system determines this frequency. Recognizing that not all operations require the same delay, some improvement in system performance can be achieved by matching the micro-cycle length to the operation being executed. In the DRAFT architecture, the fact that the effective wordlength varies from instruction to instruction means that the ability to vary the micro-cycle length is even more important. Using conventional look-ahead carry circuits, an additional delay of 30-45 ns is added for each level in the look-ahead tree. In the prototype version of the DRAFT machine, the machine will run slightly faster when split into 8 32-bit segments than when it is operating on 256-bit operands. For example, a 32-bit register-to-register add will take 166ns, while the same operation on 256-bit operands will take 200ns.

In the DRAFT prototype the micro-cycle length field controls an AMD 2925 clock generator that divides a crystal frequency of 30 mhz into clock waveforms ranging from 100 ns period (000 input) to 300ns (111 input).

1.2 The segmentation word

We dynamically connect two adjacent slices logically by gating two signals between the slices. Carry-out from the low-order slice is gated through carry look ahead circuitry to carry-in of the high-order slice. Similarly, shift/rotate-high from the high-order slice is connected to shift/rotate-low of the low-order slice to provide for bidirectional shifts.

The segmentation control word, an 8-bit field in each micro-instruction, controls this gating and hence the segment configuration during each instruction of a micro-program. Each bit of the segmentation control word controls the gating between the corresponding segment and its lower-order adjacent neighbor. The segmentation bit for slice 0 controls the gating of signals between slice 0 and slice 7 to provide a circle shift. The segmentation control word is also used by the carry look-ahead circuitry to generate the carry look-ahead signals for each segment, and by the condition code multiplexer in combining the status outputs from each slice to generate the correct condition codes at the slice, segment, and global levels.

1.3 The ALU slices

Each micro-instruction contains 8 ALU control words, one for each slice. The ALU slices in the prototype are being be built from AMD 2903 four-bit slice chips. The 2903's offer 16 general purpose registers (R0-R15), all standard ALU and shifter functions for 2's complement binary arithmetic, and a special set of 8 instructions for accelerating integer multiply and divide operations.

ALU operands are selected from the following:

| | |
|---|---|
| R0-R15 | - one of the sixteen general-purpose registers |
| Q | - the Q register (multiply accumulator) the 2903 |
| D0-D15 | - one of the sixteen locations in the current page of data memory |
| DL0-DL15 | - one of the sixteen locations in the current page of data memory from the next lower slice |

We included the DLn option so that we could have a fast 32-bit shift for arithmetic normalization and other horizontal data movement operations.

Each slice can independently address locations within the current data page. Since the data page is determined by a single 12-bit field in the control word portion of the micro-instruction, all data store operands in a micro-instruction must be in the same page. Because of the structure of the 2903, at most one of the two operands in an instruction can be from the data store. In other words, only one D or DL operand is permitted in an operation for each slice. Because it is possible that one slice might use a D operand while its neighbor uses a DL operand, yielding the same data store address, we have included arbitration circuitry so that DL operands are read-only.

Table 2 lists the ALU arithmetic and shift operations. The second field in the ALU word for each slice selects the operation for that slice.

The third field in each ALU control word contains control masks that determine how to generate a local condition code from the four status bits output by the 2903's. The ALU chips generate status outputs for zero, carry, 2's complement overflow, and sign. Table 3 shows how the masks select the appropriate conditions. Each slice produces a local signal at the end of each operation that indicates whether the result satisfied the condition dictated by the mask for that slice. The control unit latches the local condition codes from each slice and combines them to form a global condition code for transfers of control, as described in Section 1.1 above.

The latched local condition codes are returned to the slices so that these codes can be used to control conditional execution of operations in ensuing instructions. Each ALU word contains a conditional execution bit. If set, the ALU operation executes only if the local condition code is false. When the condition code is true, the slice ignores the current operation code and enters a wait state for the duration of the micro-step.

Conditional execution by each slice loosens considerably the restrictions imposed by the single micro-program control structure. Different slices can simultaneously execute instruction sequences that have little in common with each other.

## 2.0 The DRAFT micro-assembler

Microinstructions in the DRAFT machine are 237 bits long and contain 30 fields. A conventional columnar micro-assembler format would not be practical. In practice, the segmentation of the ALU means that many of the ALU instruction words are the same from one slice to another. The DRAFT micro-assembler format is designed to take advantage of this redundancy.

In the DRAFT micro-assembler language, each micro-instruction consists of two parts, which are in separate columns in assembler source programs. One part is the micro-sequencer operation and optional label and branch address fields. The sequencer instruction is specified in three fields in the righthand column of the source program. An entry in the sequencer control column means that the source line is a micro-instruction and the assembler generates a complete control word. A vertical bar (|) separates the ALU control column on the left from the sequencer control column.

The left-hand column contains the ALU control fields. The assembler uses the current settings of several pseudo-registers to determine how to fill in the ALU control fields of the micro-instruction. In the current version of the assembler there are five such pseudo-registers: the segmentation register, the ALU operation register, the shifter operation register, the condition mask register, and the data RAM page register. The contents of these registers are set by command statements inserted between micro-instructions. Once set, a register keeps its contents until another statement changes it.

The following describes details of the current version of the micro-assembler. Because we have found that programs for DRAFT tend to contain many blocks of code that are identical except for differences in segmentation, we are now adding a macro-assembly capability to the micro-assembler.

### 2.1 Micro-assembler pseudo registers

#### 2.1.1 Segmentation register, the SEG statement

DRAFT machine segments are built by combining 32-bit ALU slices designated by bits set in the segmentation control word of the control register. A bit set to 1 connects two adjacent slices, a 0 means the slices are not connected. In the DRAFT micro-assembler format the SEG statement controls the contents of the segmentation register:

SEG w1 [w2 [w3 [w4 [w5 [w6 [w7 [w8]]]]]]]

Wn = width of segment n

For example,

```
SEG  64 32 32 128   (four segments of lengths
                     64, 32,32, and 128 bits)
```

The OP, SHIFT, and COND statements described below must contain exactly one operation code or mask for each segment. Therefore, the number of operands in the SEG statement determines the number of operands in later occurrences of these other statements. Obvious restrictions on the SEG statement are that segment lengths must be multiples of 32 and the total of the lengths of the segments cannot exceed 256.

#### 2.1.2 The DATAPAGE statement

The ALU instruction word for each slice contains two 4-bit fields that select operand addresses for the ALU. These fields may specify either an ALU register (R0-R15) or a location within a 16-word page of data primary memory (D0-D15 or DL0-DL15). If a data memory address is specified, each of the slices provides a 4-bit address generated from these fields. The high-order 12-bit field of the data RAM address is shared by all segments and is taken from the data RAM page register. The DATAPAGE statement sets this hardware register as follows:

```
DATAPAGE  (12 bit address constant)
as in

DATAPAGE  1E0H

or

DATAPAGE  TABLE1
```

For all micro-sequencer instructions that can cause a transfer of control (all except EX), the same 12-bit hardware field contains a direct branch address operand for micro-program transfers of control. The micro-assembler loads the contents of the field from the address operand of the sequencer instruction. The contents of the data RAM page pseudo-register is ignored. The micro-programmer may not combine in a single instruction a microprogram transfer of control with an attempt to set the DATAPAGE register. Because the data RAM page register is also a physical register in the machine, micro-program transfers leave the previous setting of this register unchanged. Operations within a previously established data page can execute concurrently with transfers of control.

#### 2.1.3 ALU operation register, the OP statement

For each of the ALU segments an operation code and two ALU operands must be specified in the ALU OP pseudo register. The ALUs operate as a two address architecture with the result being stored at the operand A address. A single ALU instruction specification takes the form:

```
OPCODE     Operand A,Operand B
```

For example we might have the following:

```
ADD     R1,R2  (add r1 and r2, result in R1)
SUB     R2,D14 (add r2 and d14, result in r2)
MOV     R0,DL0 (transfer low order slice d0 to R0)
```

TABLE 1. SEQUENCER OPERATION CODES

| MNEMONIC | OPERATION |
|---|---|
| JZ | Jump unconditionally to location 0 |
| JSB | Jump to subroutine |
| JSBAND | Jump to subroutine on AND condition |
| JSBOR | Jump to subroutine on OR condition |
| JP | Unconditional jump |
| JPAND | Conditional jump on AND condition |
| JPOR | Conditional jump on OR condition |
| SLAND | Selector on AND condition |
| SLOR | Selector on OR condition |
| SSBAND | Select subroutine on AND condition |
| SSBOR | Select subroutine on OR condition |
| PUSH | Push micro-instruction counter |
| LOOPAND | Loop initialize on AND condition |
| LOOPOR | Loop initialize on OR condition |
| ENDCNT | End of counter loop (dec and pop) |
| ENDCJP | End counter loop and jump |
| ENDAND | End loop on AND condition |
| ENDOR | End loop on OR condition |
| EXIT | Exit loop |
| EXITAND | Exit loop on AND condition |
| EXITOR | Exit loop on OR condition |
| RPTDIR | Repeat loop direct until counter=0 |
| HALT | Halt |
| CHLTAND | Halt on AND condition |
| CHLTOR | Halt on OR condition |
| RET | Return from subroutine |
| RETAND | Return on AND condition |
| RETOR | Return on OR condition |
| LDCNT | Load counter |
| EX | Continue sequential execution |

The number of ALU instructions specified in an OP statement depends on the number of segments as dictated in the preceding SEG statement. The complete OP statement consists of the keyword OP followed by one ALU operation code and operand pair per segment.

```
OP   I1 [I2 [I3 [I4 [I5 [I6 [I7 [I8]]]]]]]
```

where
In = ALU operation code/operand pair for segment n as in

```
OP ADD  R1,R2   SUB  R2,R1  ZERO R2 (three segments)
OP TEST R1,R2   ADD  R1,R3  TEST R1,R4  CMOV R1,R5
                                        (four segments)
```

Settings for the OP pseudo register do not persist longer than one micro-instruction. Once used to generate an instruction, the assembler automatically sets the register to NOP for all segments.

Before applying an operation in a micro-instruction, each ALU tests its conditional operation bit in its local control word. If this bit is set, tha operation is carried out only if the local condition code bit indicates that the last operation that latched the local condition code bit produced a false result. The mnemonics for conditional execution are formed from those in Table 2 by appending a C to the beginning of the mnemonic, as in CADD or CNAND.

2.1.4 Shifter operation register, the SHIFT statement

Before being written into the operand A location, ALU results pass through a shifter circuit that allows single- and double-length shifts of the result before it is written. Both arithmetic and logical shifts are allowed. Double length shifts are achieved by concatenating the result with the current contents of the Q register of the ALU.

```
SHIFT   S1 [S2 [S3 [S4 [S5 [S6 [S7 [S8]]]]]]]
```

Sn = shifter operation code for segment n

For example, we could have

```
SHIFT  SLA  SLA  NOP (three segments)
SHIFT  DLL  SLL  DLL  DLL    (four segments)
```

Not all ALU operations can be combined with shifter operations in a single micro-instruction. The extended instruction set of the 2903 has built-in shifter operations; these override any micro-program settings. Like the ALU operation, the SHIFT pseudo register setting lasts for only a single instruction after which it returns to its default setting of NOP.

2.1.5 Local condition codes, the COND statement

Each ALU slice has a local condition bit that is set to true or false by a mask and test operation on the four ALU status outputs - zero, carry, sign, and overflow. The COND statment controls the latching of ALU status for each slice and sets the mask for the ALU status outputs. The COND statment specifies a condition keyword for each segment as follows:

```
COND   C1 [C2 [C3 [C4 [C5 [C6 [C7 [C8]]]]]]]
```

Cn = condition keyword for segment n

The valid condition keywords are shown in Table 3.

We could have

```
COND  POSITIVE  POSITIVE  ZERO
COND  ZERO      NONE      CARRY
```

The NONE condition, the default for the COND pseudo-register, is a no-latch operation that leaves the local condition code unchanged for that segment. Micro-instructions for which no COND statment are specified to have no effect on the local condition codes.

TABLE 2. ALU OPERATION CODES

| MNEMONIC | OPERATION | |
|---|---|---|
| Arithmetic and logic operations | | |
| SUB | A <- A-B | Subtract |
| TEST | A-B | Compare (sub w/o latch) |
| SWUB | A <- B-A | Swap and subtract |
| ADD | A <- B+A | Add |
| NOP | A <- A | No operation |
| INCA | A <- A+1 | Increment A input |
| CMPA | A <- not A | Ones complement A |
| NEGA | A <- -A | Twos complement A |
| INCB | A <- B+1 | Increment B input |
| MOV | A <- B | Data transfer |
| CMPB | A <- not B | Ones complement B |
| NEGB | A <- -B | Twos complement B |
| ZERO | A <- 0 | Zero A |
| CAND | A <- A and not B | And complement |
| XNOR | A <- A exc nor B | Exclusive nor |
| XOR | A <- A xor B | Exclusive or |
| AND | A <- A and B | And |
| NOR | A <- A nor B | Nor |
| NAND | A <- A nand B | Nand |
| OR | A <- A or B | Or |
| Shift operations | | |
| RA | Shift result right arithmetic | |
| RL | Shift result right logical | |
| DRA | Shift result double right arithmetic | |
| DRL | Shift result double right logical | |
| NOPR | No shift, result <- one of R0..R15 | |
| NOPQ | No shift, result <- Q | |
| LA | Shift result left arithmetic | |
| LL | Shift result left logical | |
| DLA | Shift result double left arithmetic | |
| DLL | Shift result double left logical | |
| EXT | Extend sign of result | |
| Special multiply/divide operations | | |
| UMUL | Unsigned multiply step | |
| SMUL1 | Twos complement multiply step | |
| INC2 | A <- A+2 | |
| SM2C | Sign magnitude twos comp. convert | |
| SMUL2 | Twos complement multiply last step | |
| SNORM | Single precision normalize (Q) | |
| DNORM | Double length normalize | |
| DIV2 | Intermediate divide step | |
| DIV3 | Final divide step | |

## 2.2 Sequencer instruction fields

In the two-column format of the DRAFT micro-assembler, pseudo-register statements are listed down the left side of the page. None of these statements actually generates a control word. They specify how certain fields of the next control word are to be completed. Generating a control word requires that a micro- sequencer instruction be specified on the right side of the page. These instructions are specified in a three-field format familiar to any assembly language programmer.

[label] operation code [address operand]

## 2.3 Comments

The line scanner terminates when it encounters a ; in the input line. If room permits documentation can be added to the right of each statement or complete lines of documentation may be added by entering a leading semicolon.

## 2.4 Examples

We now present an example of a DRAFT micro-assembler program segment. In this example the machine is segmented into two 128-bit machines each with two values stored in R1 and R2. The program segment examines the values and arranges them such that the smaller of the values ends up in R1. The following is the assembler listing:

```
                ALU CONTROL                          SEQUENCER CONTROL

    ; if r1>r2 swap r1,r2
1   seg  128          128
2   op   test r2,r1   test r2,r1
3   cond positive     positive     |0000          ex
4   op   Cmov r3,r1   Cmov r3,r1   |0001          jpand done
5   op   mov r1,r2    Cmov r1,r2   |0002          ex
6   op   Cmov r2,r3   Cmov r3,r2   |0003          ex
7                                  |0004 done     ex
```

In the example, statement 1 sets the value of the segmentation pseudo-register such that two 128-bit machines are formed from the 256-bit word. This statement generates a bit string in the segmentation control word of the micro-instruction such that slices 0,1,2,3 are combined into one segment and slices 4,5,6,7 form the second. The value for the segmentation control word is automatically included in all micro-instructions until a new SEG statement is encountered in the source listing. The number of operands in the SEG statement also determines the number of operands in later OP and COND statements.

Statement 2 assigns operation codes to each of the ALU segments. In this case a TEST operation (subtract without result store) is done in both segments.

All ALU operations generate status outputs, but not every statement is allowed to change the local condition code register. The COND statement selects a particular condition output from the ALU segment and causes that condition to be

TABLE 3. CONDITION CODE CONTROL FIELD

| MNEMONIC | MASK |
|---|---|
| ZERO | 0 0 0 1 |
| NOTZERO | 0 0 0 0 |
| POSITIVE | 0 0 1 0 |
| NEGATIVE | 0 0 1 1 |
| OVERFLOW | 0 1 0 1 |
| NOOVERFLOW | 0 1 0 0 |
| CARRY | 0 1 1 1 |
| NOTCARRY | 0 1 1 0 |
| TRUE | 1 x x 1 |
| FALSE | 1 x x 0 |

latched into the local condition code. This condition remains in effect until the next statement generated with a COND statement included. In statement 3 the condition positive (R2>R1) is latched into the local condition code of each statement.

Statement 3 is the first that generates a micro- instruction word. The EX sequencer operation simply causes the sequencer to go on to the next sequential micro-instruction after the ALU is finished.

The sequencer portion of statement 4 (JPAND) tests to see if both segments have latched a true condition from statement 3, the preceding micro-instruction that latched the condition code. If so, the sequencer jumps to DONE, because the contents of neither register pair are to be interchanged. If at least one interchange is to be performed, statements 4-6 do the interchange. Note that, because the Cmov ALU operation is used, only the segment(s) which latched a false condition in the TEST instruction are interchanged.

## 3.0 Performance estimates

Table 4 shows estimated execution times for the arithmetic operations, assuming we use a 30Mhz clock with the variable cycle-time sequencer as described above. The jump in add and subtract times that occurs between 64- and 96-bit operands results from the need to go through an additional carry-look-ahead level for operands that are 96 bits or longer. This change is also reflected in the multiply and divide times, in which the time per additional 32 bits of operand increases from 5.3 to 6.4 and 6.4 to 7.5 microseconds, respectively.

We have yet to explore fully all the ways in which the DRAFT architecture will speed up integer arithmetic. The following summarizes the results we know to date.

The DRAFT architecture has two major effects on the speed at which integer arithmetic can be done. The first derives from the long wordlength. Let us take as a competitor a computer with a 32-bit word and the same execution times as in the first column of Table 4. A 256-bit add on 32-bit machines is done in software by computing pairwise sums of 8 32-bit slices of the operands, starting at the low order ends of the operands. Each addition except the one for the highest-ordered slices (if we are not concerned about the possibility of an overflow) is followed by an add of the carry bit to one of the next higher order operand slices. Thus, 15 32-bit additions are required to compute the sum. There are computers that have an add with carry operation; such machines require 8 32-bit additions to compute a 256-bit sum. Thus, DRAFT will add 256-bit numbers 8 to 15 times faster than will machines with 32 bit words and the same basic addition time.

For economic and practical reasons, the DRAFT prototype will not have special multiply-divide circuits. The machine will still outperform short wordlength machines for these operations. For example, to multiply 2 256-bit numbers to form a 512-bit product will require 51.2 microseconds on DRAFT. The same operation on a 32-bit processor requires 64 32-bit multiplies to form 64-bit partial products and 224 additions to compute the final result. The total time, assuming the same times are required for 32-bit addition and multiplication as on the DRAFT machine, is about 375 microseconds. Thus, DRAFT should be on the order of 5 times faster than 1 Mip machines for multiplication, considering the effect of the long wordlength alone.

The DRAFT machine will compute a 128-bit quotient and 128-bit remainder from a 256-bit dividend and 128-bit divisor in 59.6 microseconds. The same operation, using the algorithm in Knuth [10] on a 32-bit machine would require about 422 microseconds.

The next generation DRAFT machine will have built-in multiply hardware, which will permit the computation of 512-bit products in 3.6 microseconds and produce a corresponding improvement in speed for division.

The other factor in the DRAFT architecture that leads to an improvement in performance is its dynamic reconfigurability. Many arithmetic operations and algorithms produce partial results that continually decrease in length, as do intermediate dividends in division or the intermediate results in GCD algorithms. In other processes, such as multiplication, the intermediate results grow in length as the computations proceed. The DRAFT architecture takes advantage of this fact by allowing parallel processing of intermediate results.

Perhaps the simplest example of this capability is in division. Suppose we have 8 division problems to do, all of which involve 256-bit dividends and divisors no longer than 32 bits. The idea is to start the division process on one of the 256-bit dividends and continue until the remaining dividend is 128 bits long. Store this intermediate dividend and the associated partial quotient temporarily. Do this for the remaining 7 256-bit operands. The result is 8 128-bit intermediate dividends.

These can now be processed two at a time in DRAFT's 256-bit word, the division proceeding until the intermediate dividends are 64 bits long. The resulting 8 64-bit intermediate dividends are processed four at a time to produce 8 32-bit dividends. The division is then completed in one 32-bit divide cycle. The result is (supposing for simplicity that all 32-bit division cycles take the same time) that only 43 32-bit divide cycles are required to compute the 8 quotients, compared with the 64 divide cycles that would be needed if no parallel computations were possible.

The same reasoning applies to the computation of GCDs. Also, a similar technique produces the same improvement in execution times for multiplication and other operations in which intermediate results grow, rather than shrink, in length.

We can produce an additional improvement by noting that, in the "tree" segementatioin approach described above, not all the the slices in the ALU are busy in all steps in the algorithm. In the example described for division, there are 52 slices that are not used for 32-bit divide cycles in the process of doing 8 divisions. These slices would be

available for other operations, such as arithmetic and tests for loop control or the generation of operands for later steps, with no loss of time.

The DRAFT architecture also permits a speed increase by a "horizontal pipeline" approach, a topic that demands much further research effort. Suppose that a multiplication is to be followed immediately by division of the product. In DRAFT, a trial value for the high-order 64 bits of the product could be computed in a 64-bit multiply cycle and moved to two slices where the first steps of Knuth's algorithm [10], normalization and the calculation of a trial quotient from the highest-order portions of the dividend and divisor, could be done while the exact product is calculated. The effect is that, for problems in which this sequence of computations occurs, the DRAFT machine will permit a considerable overlap of the division and multiplication operations.

We can use the analysis above to estimate times required for factoring numbers using methods now in vogue. For example, one of the most popular and powerful factoring methods, the Pollard p-1 technique [11], involves repeatedly raising a number to prime powers, modulo the number, N, to be factored. Because the numbers are large, the best way to proceed is to follow each multiplication with a division by N to compute the mod function. In factoring 60-digit numbers, Buell uses the 190,000-odd primes less than 2,000,000 as exponents. Each exponentiation requires an average of 30 multiplications and divisions. Using the times in Table 4 for multiplication and division we estimate that the DRAFT machine will require about 9.25 minutes to do the multiplications and divisions. Loop control calculations and logic and generating the prime exponents from a sieve can be overlapped with the multiplications and divisions.

Buell's implementation of the method also requires that the GCD of two large numbers be calculated after each group of 500 exponentiations. A GCD requires approximately .843 ln(N) divisions, for large N, where N is the larger of the two operands [12]. We have determined empirically that the average length of the dividends in computing GCDs is one-half the length of N. Therefore, computing the GCD of 2 60-digit numbers should require about 3,470 microseconds on the DRAFT machine. Thus, the GCDs should require a total of about 2.2 minutes; the total factoring time should be approximately 11.5 minutes. This analysis includes only the raw computing times that will be required. All of the calculations can take advantage of the "tree" segmentation technique, which would reduce the time to about 7.7 minutes. The same problem on the LSU System Network Computer Center's IBM 3033 requires about 90 minutes of CPU time. As mentioned above, we can expect a factor of 10 improvement when we include multiply circuitry in the next generation of the DRAFT machine.

The reason for computing so many GCDs in the Pollard p-1 method is that the exponentiation-mod cycle takes so long on the conventional machines we are using. The speed of the DRAFT processor will enable us to wait until all the exponentiations and mods are completed before computing the single GCD at the end.

Schnorr's factoring method [13] requires about 120 (we do not yet have precise statistics on the computational requirements for this method) GCDs of pairs of numbers that are of magnitude on the order of the square root of the number to be factored for each of the primes used in generating trial factors.

The method also requires the generation of a positive definite binary quadratic form, which requires computing the product of 3 2 by 2 matrices, for each prime. Assuming again that we use the 190,000-odd primes less than 2,000,000, Schnorr's method should require about 325 minutes to factor a 60-digit number, or 233 minutes with tree segmentation. We have just begun to analyze this method with the DRAFT architecture in mind. One immediate improvement would be to use DRAFT's reconfigurability to do all 8 multiplications in a matrix product in one multiplication operation. The four additions could also be done in parallel.

Finally, we consider the continued fraction method of Morrison and Brillhart [14]. This technique requires about $2.36 \times 10^{11}$ divisions of numbers about 60 digits in length to factor a 60-digit number[15]. DRAFT should do this in about 3,415 hours (2,294 hours with tree segmentation), a result that clearly indicates that there is more to these problems than choosing a fast processor.

## 4.0 Conclusion

The DRAFT machine will have a unique dynamically reconfigurable ALU, which will provide a great deal of power and flexibility for research in algorithms for factoring numbers and other problems in experimental number theory. Much interesting research remains to be done, including a formal

TABLE 4. ARITHMETIC OPERATION TIMES (usec)

| | Length of longest segment (bits) | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 32 | 64 | 96 | 128 | 160 | 192 | 224 | 256 |
| ADD | .166 | .166 | .200 | .200 | .200 | .200 | .200 | .200 |
| SUBTRACT | .166 | .166 | .200 | .200 | .200 | .200 | .200 | .200 |
| MULTIPLY | 5.3 | 10.6 | 19.2 | 25.6 | 32.0 | 38.4 | 44.8 | 51.2 |
| DIVIDE | 6.4 | 12.8 | 23.3 | 29.8 | 37.2 | 44.7 | 52.1 | 59.6 |

study of the properties of machines like DRAFT, the development of new algorithms for arithmetic that take advantage of the machine's properties, the development of high-level languages for the machine and its descendents, and the use of the machine to explore the properties of the natural number system.

REFERENCES


[1] Gauss, C. F., Disquisiones Arithmeticae 21 Sec. 329

[2] Rudd, W. G., Duncan A. Buell and Donald M. Chiarulli, "A High Performance Factoring Machine," Proc. 11th International Symposium on Computer Architecture, Ann Arbor, Michigan, 1984, pp. 297-300.

[3] Buell, Duncan A., "Computer computation of class groups of quadratic [illegible]," Proc. 8th Manitoba Conference on Numerical Mathematics and Computing, Winnipeg, Manitoba, September, 1978.

[4] Buell, Duncan A., "The expectation of success using a Monte Carlo factoring method - some statistics on quadratic class numbers," Mathematics of Computing v. 43, 1984, 313-327.

[5] Buell, Duncan A. and Richard H. Hudson, "Solutions of certain quaternary quadratic systems," Pacific Journal of Mathematics v. 114, 1984, 23-45.

[6] Buell, Duncan A., "Some factorings from the Cunningham table," LSU Computer Science Technical Report #82-1013, 1982. The factorings are reported in Brillhart, John, D. H. Lehmer, J. L. Selfridge, Bryant Tuckerman, and S. S. Wagstaff, Jr., Factorizations of b**n+-1, b=2, 3, 5, 6, 7, 10, 11, 12 up to high powers, American Mathematical Society, Providence, RI, 1983.

[7] Kolata, Gina, "Factoring gets easier," Science v. 222, 1983, 999-1001.

[8] Smith, J. W. and S. S. Wagstaff, "An extended precision operand computer," Proc. 21st Southeastern Regional ACM Conference, 1983, 209-216.

[9] Smith, J. W. and S. S. Wagstaff, "How to crack an RSA cryptosystem," Congressus Numerantium, to appear.

[10] Knuth, D. E. The Art of Computer Programming, Vol. 2. 2nd ed., Addison-Wesley, Reading, Mass., 1981, pp. 255-261.

[11] Pollard, J. M. "Theorems on factoring and primality testing," Proc. Cambridge Phil. Soc. v. 76, 1974, 521-528.

[12] Knuth, D. E., op. cit., p. 354.

[13] Schnorr, C. P. and H. W. Lenstra, Jr., "A Monte Carlo factoring algorithm with finite storage," Mathematics of Computation, v. 43, 1984, 289-312.

[14] Morrison, M. A. and J. Brillhart, "A method of factoring and the factorization of F sub 7," Mathematics of Computation, v. 29, 1975, 183-205.

[15] Wunderlich, M. C., "Implementing the continued fraction factoring algorithm on parallel machines," 1984, unpublished preprint.