Christopher Cotropia

Processor Performance/Area Optimization

Undergraduate Honors Program
Honors Advisor: Professor Bruce Holmer
Fall/Winter 1995-96
3/6/96

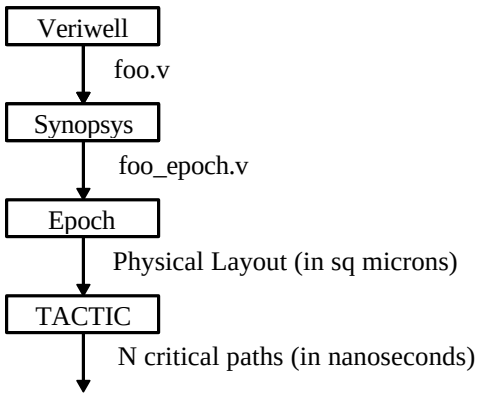
Introduction

The emphasis on Massively Parallel Processing (MPP) and other large scale multiprocessors has introduced a need to shift the way we analyze processor performance. The evolution from the 8088 to the P5 has witnessed a steady drop in performance per area, exemplifying the need to change the designer's design goals when approaching large scale, multiprocessor design.

With multiprocessors, performance and size can range over multiple orders of magnitude, and maximizing the performance for each addition in area is crucial. To assure that each dollar spent for additional processors is justified, the designer needs to maximize the performance for this additional hardware cost. Minimizing the amount of hardware for each individual processor will significantly reduce the cost of large scale multiprocessor systems.

An attempt will be made to find the design that maximizes the ratio of performance per area and the motivation that create it. Although the instruction set architecture (ISA) for a multiprocessor node would, in reality, need to be complete, trends will be studied and an attempt will be made to discover principles using simplified ISAs and benchmarks. The techniques and design decisions used will be analyzed to guide future designers in developing processors with maximizing the performance per area ratio in mind.

Methodology



Processor Design Method -

The overriding factors in obtaining a certain processor's area and performance were the ease, speed, and accuracy of the developed method. A group of compatible tools was found that would produce these metrics. Unless otherwise noted, the default switches were used in each of the CAD tools.

Every processor was initially developed using a hardware description language. Verilog was used because of its similarity to C and its compatibility with the other design tools used. Every processor designed, both accumulator and register based, were derivatives of a base Verilog file. Thus, changes between different implementations of the same base processor were simple to implement. To ease the execution of these changes, each Verilog designed processor consisted of multiple modules connected in a hierarchy. A top module attaches the smaller blocks of the processor, such as the ALU, register file or program counter. Crucial features of the Verilog code were the use of a case statement for the control logic. A recurring change made to each processor was the addition and removal of instructions. Although this type of change includes changes to other modules in the Verilog code, the control logic always needs to change, and adding or modifying a case is a simple way to accomplish this. Veriwell, a Verilog compiler and simulator, was used to test that the syntax of the Verilog code was correct and to verify the correctness of the processor design.

The next step in the design process was to use Synopsys, which is a tool that compiles the Verilog code into a hardware implementation. The Verilog code simulated in Veriwell was behavioral Verilog. It simulated the behavior of the designed processor. When the behavioral Verilog is processed by Synopsis, structural Verilog is produced. Structural Verilog uses modules that represent individual cells in a standard cell library. The new Verilog file, consisting of only structural Verilog, contained an exclusively hardware implementation of the processor. After a first pass through Synopsys, extra muxes and registers were exposed and the original Verilog code was modified. The modified code was then passed through Synopsys. One item Synopsis did not convert from behavioral to structural Verilog was the predefined module used in the register based designs, dualram. Using standard cells from the register file based ISA results in a large chip area. This was reduced by adding some structural Verilog, by hand, to the behavioral Verilog simulated in Veriwell. Essentially a predefined RAM module, dualram, is was and will eventually be implemented in hardware as RAM that is optimized for layout area.

The structural Verilog code produced by Synopsys was then used as input for Epoch, a physical layout tool. Epoch takes Verilog code and produces a wire layout of the processor on a "chip". Epoch uses predefined, standard cells, to automate the VLSI layout of the chip. The cells consist of logic gates (AND, OR, ...) and 1-bit registers. The cells required by the design are placed in rows with interconnections routed between rows. The ruleset for Epoch was a 2 micron CMOS process, the mos2u2m1pNWORB Epoch library techfile. After inputting the compiled Verilog code, and the Synopsys buffer size switch is on, the output is modified into a netlist input. The netlist of the processor is then sent through Epoch's automatic compiler to produce the physical design. Epoch then allows the user to view the physical design, and it provides the microns.

Another tool in the Epoch package, called TACTIC, determines and times the critical path of the physical design built by Epoch's automatic compile. After defining which net is the processor's clock, TACTIC builds a network of critical paths for a given design and provides the time a 0 or 1 takes to propagate from a registered output to a clocked input. No specifications were given for time delays from off processor inputs. The result was in nanoseconds and was used as the designed processor's cycle time.

Performance Metric Development -

The next question faced was what is a "good" performance metric for comparing the processors. Cycle time was not a proper metric because it may take one processor 20 cycles to complete a task that takes only another 5 cycles. Benchmarks also were developed, and each of these benchmarks were weighted to show their true importance in the performance of a processor.

To find the proper weight of each benchmark, a group of programs that performed the common task a processor performs were defined. The programs chosen completed tasks that an ASIC would have a high probability of doing. These tasks were compression (gzip), decompression (gunzip), JPEG decompression (djpeg), searching (grep), and lexical interpretation (flex). Compression and decompression of both data and images are needed in many applications of digital equipment, such as cellular phones, video relay units, and data transmission. Analysis and processing of text files can be found in such items as speech processors and database search engines. Each of these programs were compiled on six different workstations (i486/DX2 50, SPARC IPC, SPARC 5, HP 712/80, HP 715/50, and HP 715/33) and their execution times, using a common set of files for input, were recorded for five independent trials.

The next step was to define a set of common benchmarks to run on both the workstations and the newly designed processors. A group of benchmarks were chosen in a similar method to that used to pick the timed programs. Five benchmarks were written, in C for the workstations and written in assembly for the designed processors. They were compiled on each machine using GCC with the optimization option on. These benchmarks included: insertion sort, hash searching (with chaining), multiplication by a large constant, string copy, and string compare. Each of the benchmarks perform a basic task that programs often employ. The benchmarks were then nested inside for loops to assure that their execution time would be large enough to measure when run on the six workstations. The benchmark's execution and the empty loop's execution were timed five times.

The execution time of the programs, benchmarks, and empty loops were averaged for each machine. The empty loop's averaged time was subtracted from its relative benchmarks time so that the time produced was unique to instructions in to the benchmark. A least square method was used to find the weight associated with each benchmark that would total the execution time of a given program. Using Matlab to find the non-negative least square fit (NNLS), a matrix of the machines' averages were found. Matlab provided the coefficients that met this criteria.

To ensure that the values the NNLS produced were accurate, the standard error was determined. Since the system was complex, and the variables involved were unknown, the bootstrap method of standard error determination was used.[*] This involved automation of the least square fit process and the random selection of the timed execution of benchmarks and programs. The five execution times taken for each benchmark and program on each machine were randomly picked to create a new set of five execution times (which could contain duplicates) for the respective benchmark or program. These new sets of five values were averaged and the coefficients were evaluated. The random selection to evaluate execution times was repeated 2000 times. The standard deviation between these 2000 values and the average of each benchmark weight was then evaluated.

Programs	Gzip		Gunzip		Flex		Djpeg		Grep	
Benchmarks	weight	error	weight	error	weight	error	weight	error	weight	error
Hash Search	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Insertion Sort	0.0241	0.0	0.0067	0.0	0.0010	0.0	0.0300	0.0	0.0003	0.0
Multiplication	5.2317	0.0005	0.3604	0.0012	0.0443	0.0012	4.4364	0.0010	0.0318	0.0007
String Cmp	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
String Copy	0.0101	0.0001	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Performance Metric Results -

The results of the non-negative least square fit are interesting and worth investigating. When determining the weights that were going to be attributed to each benchmark, only three received weights. The others (hash searching, string compare) had a weight of zero. Even the standard deviation, taken to 4 significant digits, from the bootstrap method was zero for these benchmarks. Insertion sort, multiplication by a constant, and string copy were the only ones receiving weights and since string copy only received weight for gzip it is not worth noting.

Multiplication by a constant consistently received the highest weight for each program. The most reasonable explanation for this is that the multiplication benchmark is basically measuring cycle time. The benchmark includes shift lefts and adds, both of which are probably one of the critical paths for the tested workstations, and the resulting execution time displayed the cycle time of the workstations processor. Every program includes instruction that will take only one cycle which are represented by the multiplication benchmark. The weight attributed to this benchmark demonstrates the need for a process designer to reduce the single cycle time as low as possible, and to have as many instruction as possible be executed in a single cycle.

[*] Bootstrap Method of error determination is described in Bradley Efron and Robert J. Tibshirani book An Introduction to the Bootstrap (Chapman & Hall, 1993)

The insertion sort benchmark exposes the time it takes for the processor to read and write from memory, and the time it takes for the processor to branch. The building blocks of the benchmark are loads and stores from the list, as well as jumps based on the comparison of these values. Memory references and

branching are two tasks that usually take more then one cycle for a processor to perform. Reducing the number of cycles to carry out these instructions greatly increase the performance of the processor.

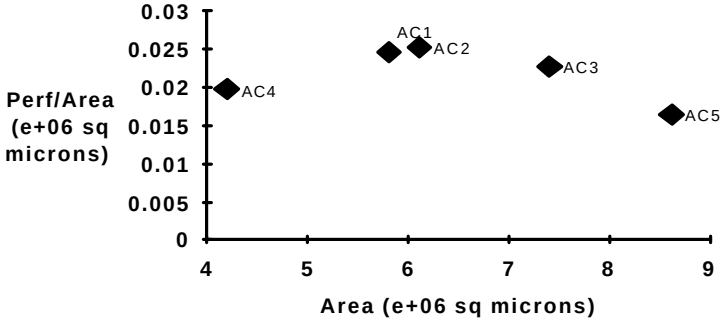
Combining the instructions included in these two benchmarks provides the basic instructions a processor needs to execute. Optimizing a processor to execute these two benchmarks quickly, will increase the performance of the basic tasks of a processor. These changes will speed up the processors execution of any program.

Results

Processor design evaluation -

There where two basic designs created, an accumulator based and a register file based design. Five derivatives of each of these two basic designs were evaluated, and size and performance values were found. Each of these two base designs and their derivatives had unique advantages in speed and size, and some derivatives did not gain anything from the additional area or lower cycle time they acquired.

Some basics were common between both base designs. A 16-bit datapath was implemented in both processors, and a 16-bit instruction path was also implemented unless otherwise noted. The instruction memory was separate from data memory. The ALU only performed addition. [Subtraction was done by inverting the subtracted input and setting the carry in to one.] Incrementing was also done by setting one of the ALU's inputs at one. A left shifter was placed in a module external to the ALU and shifted by 1 bit, unless otherwise noted. The program counter was self incrementing in all designs with a parallel load for jumps. An additional assumption was, made so reads and writes only took one clock cycle, and the control logic for stores and loads also implement this assumption. Instruction that do not include a memory read or write take only one cycle. Those that do include the memory take two.



Design	Area (e+06 sq microns)	Performance/Area (e+06 sq microns)
AC1	5.81	0.024685
AC2	6.11	0.025309
AC3	7.41	0.022807
AC4	4.2	0.019731
AC5	8.62	0.016574

Accumulator Based -

The first set of processor designs are accumulator based. These processors do not include a register file but two registers referred to as accumulators. All of the instructions this design can execute involve the

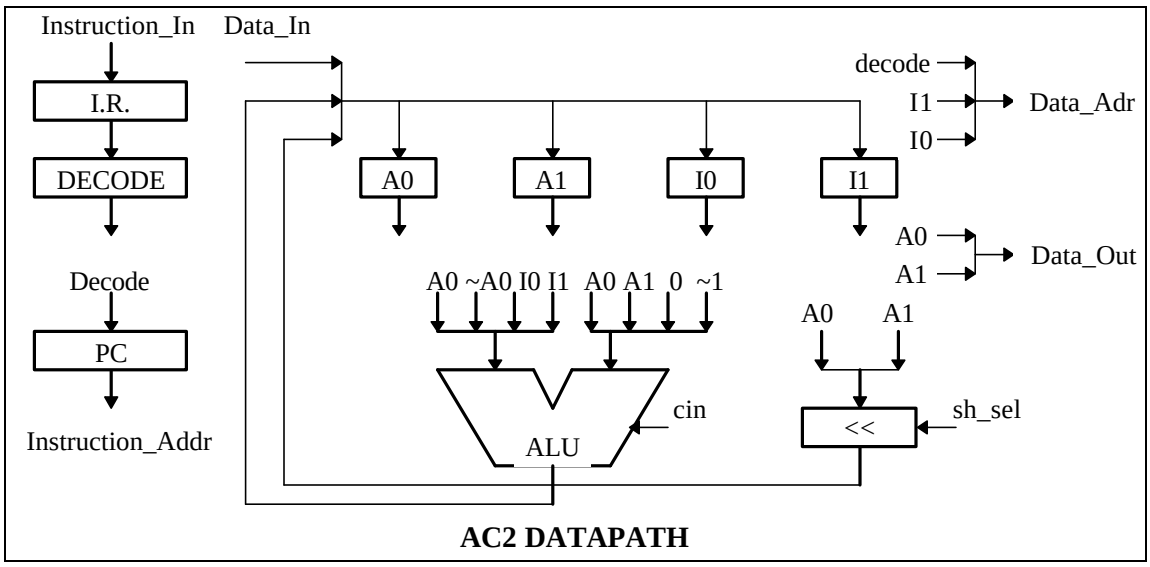
manipulation of data between the two accumulators. Adds, subtractions, and jumps are all based on the values in these accumulator registers. To aid in memory addressing, index registers were added. They are used for index addressing and are incremented and decremented through the ALU.

A few of derivatives of the design were created. AC1 (derivative 1 of the accumulator design) is the basic design. No changes were made from the description noted above. AC2 included a shift left by 2, and a modification to the shifter to allow this instruction. AC3 is similar to AC2 with new jump instruction included. Comparison logic was added in AC3 that allows for jump less than, jump equal, or jump not equal to occur in one cycle instead of two separate instructions as in AC1 and AC3 (such as a subtract and a jump if zero for jump equal). The next derivative, AC4, contains only one accumulator and one index register with a shift left by two instruction. AC5 is exactly like AC3 except the incrementing and decrementing of the index registers is internal to the register instead of going through the ALU.

Specific advantages and disadvantages appeared from the accumulator based design and its derivatives. The base design had a unique advantage in that most of the benchmarks only needed two registers. Therefore, extra registers that would take up space in a register file based design, are not present. String copy and insertion sort involve the comparison, loading, and storing of two elements at a time. Moving through the two memory areas in the string copy benchmark was easy with the use of the two index registers. Additionally, the multiplication by a constant only needed registers for the multiplicand and the product. The two accumulator based design could perform the string copy and multiplication benchmark in the same amount of instructions as the respective register file based design without the extra area the register file takes up.

The only area in which the base accumulator design lost ground to the register file design was in the number of cycles it took to perform the insertion sort benchmark. The register file, with its ability to emulate more then two index registers, was able to handle the multiple index addressing present in swapping data elements when sorting. The accumulator design only had two index registers and needed to load and store the indexes, which took a significant number of extra cycles.

The addition of instructions that combined two instructions, such as shift left by 2 or jump not equal, proved to be useful in increasing the performance. Addition of a shift left by 2 instruction added some additional area (3.0e+05 sq microns between AC1 and AC2), but also lowered the cycle time for the multiplication by constant benchmark from 11 cycles to 8. Since this benchmark consistently received the most weight in all of the programs, lowering the time to execute proved fruitful here. The addition of combined compares and jumps also lowered the number of cycles to perform the insertion sort and string compare. Although extra area was needed, this area helped to boost performance but did not improve the performance per area ratio. An important point is that these instructions did not increase the cycle time. If the cycle time was increased, these additions could possibly decrease performance and the additional area would not be useful.

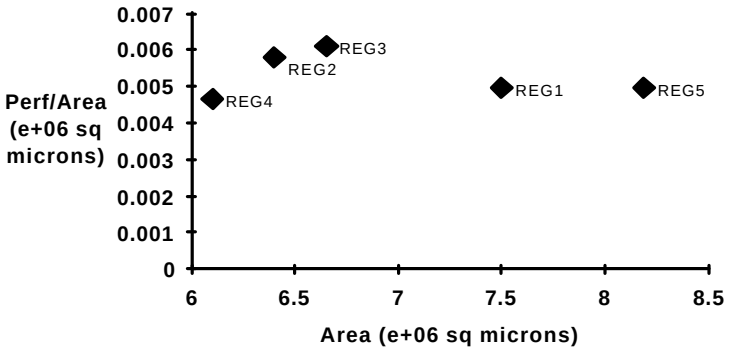


AC2 DATAPATH

Design changes present in AC4 and AC5 proved to be disadvantageous. The use of a single accumulator and index register in AC4, although decreasing the processor area significantly (to 4.2+06 sq microns),

increased the number of cycles it took to perform each benchmark. Too many loads and stores, which take two cycles, occur since each benchmark involves consistent manipulation of more then one piece of data. AC5 included self-incrementing and decrementing index registers, though thought to possibly decrease cycle time, actually increased both the size and cycle time.

The design that maximized the performance per area ratio for the accumulator based designs was AC2. AC2 has just the right amount of accumulators to optimize the code for all three benchmark. AC2's ISA also included a shift left by 2, which significantly sped up the execution of the multiplication benchmark. AC4 did not have enough area to perform efficiently, since the three benchmarks involved the constant manipulation of two pieces of data and a single accumulator prompted to many memory references. AC1 did not contain enough instructions, lacking a shift left by 2, to use its area efficiently. The addition of too many instructions proved to be harmful in AC3's design. The addition of combined compares and jumps sped up execution time but also increased area and reduced the performance per area ratio. AC5 had additions, self-incrementing and decrementing index registers, that added area but did not increase performance.



Design	Area (e+06 sq microns)	Performance/Area (e+06 sq microns)
REG1	7.5	0.004937
REG2	6.4	0.005799
REG3	6.65	0.006094
REG4	6.1	0.004651
REG5	8.19	0.004939

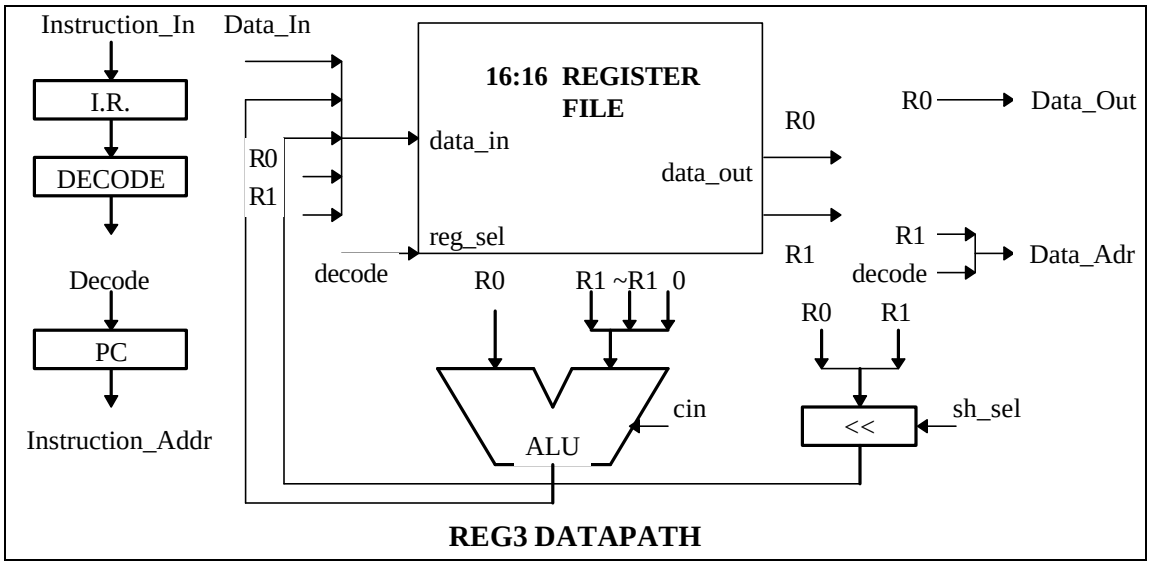
Register File Based -

The second set of designed processors are register file based. Instead of having only two registers and two index registers like the accumulator based model, a collection of 32 registers is present. Any of these registers can be used in instructions, and any memory reference can be indexed by any of these registers contents. The register has one input and two outputs, with one of the registers capable of being read from and written to in the same cycle. To aid in specifying the two registers to be used in the instruction, a 20-bit instruction was used.

Five derivatives to this base design were created. REG1 was the base design consisting of a 20-bit instruction word and 32 registers. REG2 reduced the number of registers to 16 and the instruction size to 16-bit. REG3 is similar to REG2 except that a shift left by 2 instruction was added and the shifter changed to accommodate this change. REG4 reduced the register file size to 8 and included the shift left by 2 instruction. REG5 increased the register file size to 64 and returned to the 2-bit instruction.

The main disadvantage of the register file based design was the slow cycle time the use of the dualram datapath module in Epoch produced. Explicitly defining a register file in Verilog, although producing a reasonable cycle time, produced a significantly larger processor and a lower performance per area ratio. The use of the datapath module for the register file gave similar area sizes to the accumulator based designs, but increased the cycle time four fold (from 88.92ns for AC1 to 402.558ns for REG1). The large cycle time hampered any advantages the register file had over the accumulator based design. The basic

advantage is the lower cycle times for benchmarks that needed more then two registers to operate. This was only the case for index addressing in the insertion sort benchmark. Since the storing or loading takes an extra cycle, anytime one can stay on chip, it speeds up execution significantly.



REG3 DATAPATH

The same advantages to certain derivatives of the accumulator based designs appear here in the register file design. The addition of the shift left by 2 instruction in REG3 improved performance for the same reasons it helped AC2 and AC3. Additionally, only 13 registers can be used to reduce the cycle time in the insertion sort benchmark. So, any register file that is greater then 16, found in AC1 and AC5, only adds additional area and provides no performance increase. Any register file with less then 16, found in AC4, increases the cycle count to perform an insertion sort. Both of these design changes do not improve the performance per area ratio, but in fact hamper it.

The design that maximized the performance per area ratio for the register based designs was REG3. REG3 has just the right amount of registers to optimize the code for the insertion sort benchmark. REG3's ISA also included a shift left by 2, which significantly sped up the execution of the multiplication benchmark. REG4 did not have enough area to perform efficiently since 8 registers caused it to have to

many memory references in the insertion sort benchmark. REG1 and REG5 contained an excess of unused area, 32 and 64 registers respectively. REG2 had the adequate number of registers, but its ALU did not contain a shift left by 2, to help optimize the processor's execution of the multiplication benchmark.

Conclusion

A question arises if a two cycle wait is a proper model for the simulation of reads and writes from memory. For simplicity reasons, two cycles were the easiest model to implement and still install an extra cycle for instructions that read or wrote to memory. So, benefits to using data already present on chip were preserved. Additionally, a quick analysis using a model of a 250ns wait does not change the results drastically. The main differences in the results are an increase in the performance per area ratio for REG4, and a decrease in the performance per area ratio for AC4. Since REG4 has a smaller register set (8 register instead of the standard 16), REG4's benchmarks needed more memory references. But, since REG4 cycle time was 399.519ns, a two cycle load or store took about 800ns while a 250ns wait model significantly reduces the execution time of the benchmarks. The exact opposite is true of AC4. AC4 only has one accumulator, prompting the same need as REG4 for more memory references, but the cycle time of AC4 was 87.843ns. A two cycle wait for a load or store is less then 200ns, which is less then the wait in the 250ns model.

Although the performance per area ratio has probably not been truly maximized in the 10 processors developed, the basic framework on how to go about developing processors that optimize performance per area was developed. By determining which benchmarks hold the most importance in determining a processors' performance, the specific areas where a processor needs to improve are well defined (memory accesses, cycle time, and branching) and these improvements need to come with the least increase in area as possible. The advantages of certain derivatives, AC2, demonstrate that small changes, like adding a single instruction, can produce noticeable improvements. From the benchmarks, the number of registers and ALU functions that are necessary to decrease the execution time to a minimum can be clearly stated. For the multiplication, insertion, and string copy benchmarks, a processor needs 14 register and an ALU that adds, subtracts, shifts left by 1, and shifts left by 2 to minimize execution time. Combining compares and jumps will speed up the execution time of the insertion sort and string copy. But a designer must be careful: the addition of combined compares and jumps actually reduced the performance per area ratio in AC3. Just being aware of the importance of maximizing the performance per area ratio is a step in the right direction for the computer architecture designer.

Appendix A - Benchmark Code

A.1 - Benchmarks in C code -

Hash Search:

```
#include <stdlib.h>
#include <stdio.h>
#define TABLE_SIZE  128
#define MAX_CHAR 100

struct bucket {
       char strg[MAX_CHAR];
       struct bucket *next;
} hash_table[TABLE_SIZE];

unsigned int hash_value(char *key);
void str_Cpy(char *to,char *from);
char *find_hash(char *find,struct bucket ht[]);
void insert_hash(char *in_str,struct bucket ht[]);
void init_table(struct bucket ht[]);

void main (void)
{
char in_str[MAX_CHAR],found[MAX_CHAR];
unsigned int i;
unsigned long k;

/*Init hash table*/
init_table(hash_table);

insert_hash("asd",hash_table);
insert_hash("woiejfasd",hash_table);
insert_hash("as",hash_table);


for (k=0;k<100000;k++)
       find_hash("as",hash_table);


exit(0);
}

unsigned int hash_value(char *key)
{
unsigned int i,j,value;

i=j=value=0;

/*Go until end of string */
while (key[i]!='\0')
{
  /* Mutiply has_value by seven (by shift left 3 times and sub from self) */
  j = value<<1;
  j = j<<1;
  j = j<<1;
  value = j-value;
  /* Now add in next character */
value = value + key[i];

  i++;
}
```

```
return (value & 127);
}

void init_table(struct bucket ht[])
{
  int i;
  for (i=0; i< TABLE_SIZE; i++)
  {
        ht[i].strg[0] = '\0';
        ht[i].next = NULL;
  }
}


void insert_hash(char *in_str,struct bucket ht[])
{
unsigned int key;
struct bucket *next_b,*current_b;

key = (hash_value(in_str));
/*first check to see if in first struct bucket*/
if (!strcmp(in_str,ht[key].strg))
        return;
/*if not in first struct bucket, and struct bucket empty fill it*/
if (ht[key].strg[0] == '\0')
{
        str_Cpy(ht[key].strg,in_str);
        return;
}

/*now search down the struct bucket of list*/
next_b=ht[key].next;
current_b=&ht[key];

/*Go to end of struct bucket list*/
while(next_b!=NULL)
{
        if (!strcmp(in_str,next_b->strg))
                return;
        current_b=next_b;
        next_b=current_b->next;
}

/* Now add new struct bucket and fill it */
next_b = (struct bucket *) malloc (sizeof(struct bucket));
current_b->next = next_b;
next_b->next = NULL;
str_Cpy(next_b->strg,in_str);
}

char *find_hash(char *find,struct bucket ht[])
{
unsigned int key;
struct bucket * next_b;
char temp;

/*Get index where we need to start looking*/
key = (hash_value(find));
/*First check 1st bucket in array*/
if (!strcmp(find,ht[key].strg))
return (ht[key].strg);

/*Now start going down the chain of buckets*/
next_b=ht[key].next;

while(next_b!=NULL)
{
```

```
        if (!strcmp(find,next_b->strg))
                return (next_b->strg);
        next_b=next_b->next;
}

/*Didn't find it, must not be in hash table*/
temp='\0';
return (&temp);
}

void str_Cpy(char *to,char *from)
{

int i;

i=0;
to[i] = from[i];

/*Keep copy char by char until we pass NULL char*/
while (from[i]!='\0')
{
        i++;
        to[i] = from[i];
}
}
```

Insertion Sort:

```
#include <stdlib.h>

void main (void)
{

unsigned short input[10] = {14,8,0,15,34,17,3,19,0,33} ;

/* Seems algorithm implied an array, for a linked list would not need
   to mover elements down */
unsigned short sorted[10];
unsigned short i,j;
unsigned long k;

for (k=0;k<100000;k++){

/* Initially set index to 0, and put first item in sorted list */
i=0;
sorted[i] = input[i];
i++;

/* While there are elements to be sorted...i.e. i<10 */
while (i<10)
{
  /* J needed for movement */
  j=i;

  /* We start from the end of the sorted list and keep going till
     we either are at the beginning, or have placed the item */
  while (j>0)
  {
    /*Still below the spot we need to place item, move compared piece down*/
    if (input[i]<sorted[j-1])
    {
        sorted[j]=sorted[j-1];
        sorted[j-1]=input[i];
        j--;
    }
    else /*This is were it goes, also handles dups*/
```

```
    {
        sorted[j]=input[i];
        j=0;
    }
  }
  i++;
}
}

}
```

Multiplication by Constant:

```
#include <stdlib.h>
#include <stdio.h>

void main (void)
{
unsigned multiplicand,product;
unsigned long i;

for (i=0;i<100000;i++)
{
multiplicand=0xaaaa;
multiplicand = multiplicand<<2;
product = multiplicand;
multiplicand = multiplicand<<2;
product = product + multiplicand;
multiplicand = multiplicand<<1;
product = product + multiplicand;
multiplicand = multiplicand<<2;
product = product + multiplicand;
}
printf("%u\n",product);
}
```

String Compare:

```
#include <stdlib.h>

void main (void)
{

unsigned long i;
char test1[20] = "atestforme";
char test2[20] = "atestforyou";
unsigned int k,done;

for (i=0;i<100000;i++)
{
k=done=0;
while (test1[k]!='\0' || !done)
{
        if (test1[k]!=test2[k])
                done=1;
k++;
}
}
}
```

String Copy:

```
#include <stdlib.h>
#include <stdio.h>

void main(void)
{

char from[14]="atestforme";
char to[14];
int i;
unsigned long k;

for (k=0;k<100000;k++)
{

i=0;
to[i] = from[i];

/*Keep copy char by char until we pass NULL char*/
while (from[i]!='\0')
{
        i++;
        to[i] = from[i];
}
}
printf("%s\n",to);
}
```

A.2 - Weighted Benchmarks in AC2 Assembly Code -

Insertion Sort:

```
movi    i0,input_base
movi    i1,sort_base
loadi   i0,a0
storei  i1,a0
inci    i0
:Inserting
movi     i1,sort_base
load    i,a1
store   j,a1
addi    i1,a1
deci    i1
loadi   i1,a1
loadi   i0,a0
sub     a1,a0
jneg    a1,STORE
loadi   i1,a1
inci    i1
storei  i1,a1
deci    i1
storei  i1,a0
deci    i1
load    j,a0
dec     a0
store   a0,j
jnz     a0,Inserting
goto    END
STORE:
inci    i1
storei  i1,a0
END:
load    i,a0
inc     a0
inci    i0
```

```
load    a1,#ofitems_addr
sub     a1,a0
jneg    a1,Inserting
```

Multiplication by Constant:

```
shl2    a0
mova    a1,a0
shl2    a0
add     a1,a0
shl1    a0
add     a1,a0
shl2    a0
add     a1,a0
```

String Copy:

```
movi    i0,to_base_addr
movi    i1,from_base_addr
START:
loadi   a0,i1
load    a1,'\0'_addr
sub     a1,a0
jz      a1, DONE
storei  i0,a0
inci    i1
inci    i0
goto START
DONE:
```

A.3 - Weighted Benchmarks in REG3 assembly code -

Insertion Sort:

```
load    r1,sorted_base
load    r2,index_base
load    r3,addr_1
loadr   r2,r4
storer  r1,r4
INSERT:
mov     r4,r3
mov     r5,r1
add     r5,r4
mov     r6,r2
add     r6,r3
loadr   r5,r7
loadr   r5,r12
mov     r8,r5
dec     r8
loadr   r8,r10
mov     r11,r10
sub     r11,r12
jneg    r11,STORE
storer  r5,r10
storer  r8,r12
dec     r4
jnz     r4,INSERT
goto DONE
STORE:
storer  r5,r12
```

```
DONE:
inc     r3
load    r13,#elements_addr
mov     r13,r14
sub     r14,r3
jneg    r14,INSERT
```

Multiplication by Constant:

```
shl2    r1
mov     r2,r1
shl2    r1
add     r2,r1
shl1    r1
add     r2,r1
shl2    r1
add     r2,r1
```

String Copy:

```
load    r1,to_base
load    r2,from_base
START:
loadr   r2,r3
load    r4,'\0'_addr
sub     r4,r3
jz      r4,DONE
storer  r1,r3
inc     r2
inc     r1
goto    START
DONE:
```

Appendix B - Verilog Code

B.1 - AC2 Verilog Code -

```
module ac2(ir_in, ir_addrout, data_in,
                              dataaddr,clock,reset,data_out,data_we);

input clock,reset;
input [15:0] data_in;
input [15:0] ir_in;
output [15:0] dataaddr;
output [15:0] ir_addrout;
output [15:0] data_out;
output data_we;

wire [15:0] a0_out,a1_out,ac_in, i0_out,i1_out,i_in,alu_out,a_in,b_in;
wire [15:0] ir_out, shift_out;
wire [1:0] alua_sel, alub_sel, a_sel, addrout_sel;
wire [2:0] pcload_sel;
wire a0we,a1we,i0we,i1we,rw,loadpc,pcwe,index_sel,dataout_sel,changestate;
wire shift_sel;
reg state;

dataout_mux u1sub1 (a0_out,a1_out,dataout_sel, data_out);
addrout_mux u1sub2 (i0_out, i1_out, ir_out[8:0], dataaddr, addrout_sel);
ir u1sub3 (clock, ir_in,ir_out);
pcload_mux u1sub4(pcload_sel, loadpc,ir_out[10],a1_out,a0_out);
pc u1sub5 (reset, clock, pcwe,loadpc,(ir_out[8:0] | 16'h00),ir_addrout);
ac_mux u1sub6 (data_in, alu_out,shift_out,a0_out,a1_out,ir_out[10],
                                                    ac_in,a_sel);
ac u1sub7a0(clock,ac_in,a0we,a0_out);
ac u1sub7a1(clock,ac_in,a1we,a1_out);
i_mux u1sub8 (data_in, alu_out, index_sel, i_in);
index u1sub9i0 (clock,i_in,i0we,i0_out);
index u1sub9i1 (clock,i_in,i1we,i1_out);
alua_mux u1sub10 (a0_out,i0_out,i1_out,a_in,alua_sel);
alub_mux u1sub11 (a0_out,a1_out,b_in,alub_sel);
alu u1sub12 (a_in,b_in,alu_out);
shifter u1sub13 (a1_out,a0_out,shift_out,ir_out[10],shift_sel);
controller u1sub14 (ir_out, state, alua_sel, alub_sel, addrout_sel,
       index_sel, a_sel, dataout_sel,pcload_sel,a0we,a1we,i0we,i1we,rw,
       loadpc,pcwe,changestate,shift_sel);

assign data_we = rw;

always @(posedge clock)
       begin
              if (changestate)
                     state=~state;
              else
                     state=state;
       end


endmodule

module dataout_mux (a0_out,a1_out,dataout_sel, dataout); input [15:0] a0_out,a1_out;
input dataout_sel;
```

```
output [15:0] dataout;

assign dataout = (dataout_sel) ? a0_out : a1_out;

endmodule

module addrout_mux (i0_out, i1_out, ir, addrout, addrout_sel);

input [15:0] i0_out,i1_out;
input [8:0] ir;
input [1:0] addrout_sel;
output [15:0] addrout;

assign addrout = ((addrout_sel == 2'b11) ? i0_out :
                ((addrout_sel == 2'b10) ? i1_out :
                 (ir | 16'h0000)));
endmodule


//instruction register
module ir(clock, ir_in,ir_out);

input clock;
input [15:0] ir_in;
output [15:0] ir_out;

reg [15:0] ir_out;
always @(posedge clock)
       begin
              ir_out=ir_in;
       end
endmodule

module pcload_mux(pcload_sel, pcload_out,ir10,a1_out,a0_out);

input [15:0] a1_out,a0_out;
input [2:0] pcload_sel;
input ir10;

output pcload_out;

wire [15:0] a_testing;

assign a_testing = ((ir10) ? a1_out : a0_out);
assign pcload_out = ((pcload_sel == 3'b100) ? (a_testing[15]):
                      ((pcload_sel == 3'b011) ? (!a_testing):
                      ((pcload_sel == 3'b010) ? (a_testing):
                      ((pcload_sel == 3'b001) ? 1'b1:
                             (1'b0)))));

endmodule

// program counter logic
module pc(reset, clock, pcwe,loadpc,pc_in,pc_out);

input reset,clock,pcwe,loadpc;
input [15:0] pc_in;
output [15:0] pc_out;

reg [15:0] pc_out;

always @(posedge clock)
  begin
    if (reset)
        pc_out=16'h0000;
    else if (pcwe)
        if (loadpc)
```

```
                pc_out= pc_in;
        else
                pc_out=pc_out+1;
  end
endmodule

module ac_mux(data_in, alu_out,shift_out,a0_out,a1_out,ir10,ac_in,ac_sel);

input [15:0] data_in, alu_out,shift_out,a0_out,a1_out;
input [1:0] ac_sel;
input ir10;

output [15:0] ac_in;

wire [15:0] a_in;

assign a_in = ((ir10) ? a0_out : (a1_out));
assign ac_in = ((ac_sel == 2'b01) ? shift_out:
                        ((ac_sel == 2'b10) ? a_in:
                        ((ac_sel == 2'b11) ? data_in:
                          (alu_out))));
endmodule

//accumulator register
module ac(clock,ac_in,ac_we,ac_out);

input clock,ac_we;
input [15:0] ac_in;
output [15:0]ac_out;

reg[15:0] ac_out;


always @(posedge clock)
        begin
                if (ac_we)
                        ac_out=ac_in;  //if write enable high, ac=ac_in
        end
endmodule

module i_mux(data_in, alu_out, index_sel, i_in);

input [15:0] data_in,alu_out;
input index_sel;

output [15:0] i_in;

assign i_in = ((index_sel) ? data_in:
                        (alu_out));
endmodule

//index register
module index(clock,index_in,index_we,index_out);

input clock,index_we;
input [15:0] index_in;
output [15:0]index_out;

reg[15:0] index_out;
wire [15:0] index_in;

always @(posedge clock)
        begin
                if (index_we)
                        index_out=index_in;  //if write enable high, index=index_in
        end
endmodule
```

```
module alua_mux(a0_out,i0_out,i1_out,alua_out,alua_sel);

input [15:0] a0_out,i0_out,i1_out;
input [1:0] alua_sel;

output [15:0] alua_out;

assign alua_out = ((alua_sel == 2'b01) ? (~a0_out+1'b1):
                     ((alua_sel == 2'b10) ? i0_out:
                     ((alua_sel == 2'b11) ? i1_out:
                       (a0_out))));
endmodule

module alub_mux(a0_out,a1_out,alub_out,alub_sel);

input [15:0] a0_out,a1_out;
input [1:0] alub_sel;

output [15:0] alub_out;

assign alub_out = ((alub_sel == 2'b01) ? (~1'b1+1'b1):
                     ((alub_sel == 2'b10) ? 1'b1:
                     ((alub_sel == 2'b11) ? a0_out:
                       (a1_out))));
endmodule

module alu(a_in,b_in,alu_out);

input [15:0] a_in,b_in;
output [15:0] alu_out;

assign alu_out=a_in+b_in;

endmodule

module shifter (a1_out,a0_out,shift_out,ir10,shift_sel);

input [15:0] a1_out,a0_out;
input ir10,shift_sel;
output [15:0] shift_out;

assign shift_out = ((ir10 & shift_sel) ? (a1_out<<2):
                  ((~ir10 & shift_sel) ? (a0_out<<2):
                  ((ir10 & ~shift_sel) ? (a1_out<<1):
                  (a0_out<<1))));

endmodule


module controller (instruction, state, alu_asel, alub_sel, addrout_sel,
       index_sel, a_sel, dataout_sel,pcload_sel,a0we,a1we,i0we,i1we,rw,
       loadpc,pcwe,changestate,shift_sel);

input [15:0] instruction;
input state;
output [1:0] alu_asel, alub_sel, a_sel, addrout_sel;
output [2:0] pcload_sel;
output a0we,a1we,i0we,i1we,rw,loadpc,pcwe,index_sel,dataout_sel;
output changestate;
output shift_sel;

reg [1:0] alua_sel, alub_sel, a_sel, addrout_sel;
reg [2:0] pcload_sel;
reg a0we,a1we,i0we,i1we,rw,loadpc,pcwe,index_sel,dataout_sel,changestate;
reg shift_sel;
```

```
always @(instruction or state) begin

  alua_sel = 2'b00;
  alub_sel=2'b00;
  addrout_sel=2'b00;
  index_sel=1'b0;
  a_sel=2'b00;
  loadpc=1'b0;
  pcload_sel=3'b000;
  rw=1'b0;
  pcwe=1'b1;
  dataout_sel=instruction[10];
  a0we=~instruction[10];
  a1we=instruction[10];
  i0we=~instruction[9];
  i1we=instruction[9];
  changestate=1'b0;
  shift_sel=1'b0;

casex ({instruction[15:11],state})
  // loadi
  6'b000000 :
    begin
       index_sel=1'b1;
       a0we=1'b0;
       changestate=1'b1;
       pcwe=1'b0;
    end
  // loadi
  6'b000001 :
        begin
       index_sel=1'b1;
       a0we=1'b1;
       changestate=1'b1;
    end
  // loadbyi
  6'b000010 :
    begin
       a_sel=2'b11;
       if (instruction[9])
                      addrout_sel=2'b10;
       else
                      addrout_sel=2'b11;
       i0we=1'b0;
       i1we=1'b0;
           pcwe=1'b0;
           changestate=1'b1;
    end
  // loadbyi
  6'b000011 :
    begin
       a_sel=2'b11;
       if (instruction[9])
                      addrout_sel=2'b10;
       else
                      addrout_sel=2'b11;
       i0we=1'b0;
       i1we=1'b0;
       changestate=1'b1;
    end
  //storebyi
  6'b000100 :
    begin
       a0we=1'b0;
       a1we=1'b0;
           i0we=1'b0;
       i1we=1'b0;
```

```
     rw=1'b1;
     pcwe=1'b0;
        if (instruction[9])
                    addrout_sel=2'b10;
     else
                    addrout_sel=2'b11;
     changestate=1'b1;
      end
//storebyi
6'b000101 :
  begin
     a0we=1'b0;
     a1we=1'b0;
        i0we=1'b0;
     i1we=1'b0;
     rw=1'b1;
        if (instruction[9])
                    addrout_sel=2'b10;
     else
                    addrout_sel=2'b11;
        changestate=1'b1;
      end
//inci
6'b000110 :
  begin
     a0we=1'b0;
        if (instruction[9])
                    alua_sel=2'b11;
     else
                    alua_sel=2'b10;
     alub_sel=2'b10;
  end
//deci
6'b001000 :
  begin
     a0we=1'b0;
        if (instruction[9])
                    alua_sel=2'b11;
     else
                    alua_sel=2'b10;
     alub_sel=2'b01;
  end
//addi
6'b001010 :
  begin
     a0we=1'b0;
        a1we=1'b0;
        if (instruction[9])
                    alua_sel=2'b11;
     else
                    alua_sel=2'b10;
     if (!instruction[10])
                    alub_sel=2'b11;
      end
//sub
6'b001100 :
  begin
     i0we=1'b0;
     alua_sel=2'b01;
      end
//jneg
6'b001110 :
  begin
     a0we=1'b0;
     a1we=1'b0;
     i0we=1'b0;
     pcload_sel=3'b100;
```

```
     end
//jnz
6'b010000 :
  begin
     a0we=1'b0;
     a1we=1'b0;
     i0we=1'b0;
     pcload_sel=3'b010;
     end
//jz
6'b100000 :
  begin
     a0we=1'b0;
     a1we=1'b0;
     i0we=1'b0;
     pcload_sel=3'b011;
     end
//goto
6'b010110 :
  begin
     a0we=1'b0;
     a1we=1'b0;
     i0we=1'b0;
     pcload_sel=3'b001;
     end
//load
6'b010010 :
  begin
        i0we=1'b0;
        a_sel=2'b11;
        changestate=1'b1;
        pcwe=1'b0;
     end
//load
6'b010011 :
  begin
        i0we=1'b0;
        a_sel=2'b11;
        changestate=1'b1;
     end
//store
6'b010100 :
  begin
        i0we=1'b0;
        a0we=1'b0;
        a1we=1'b0;
        changestate=1'b1;
        pcwe=1'b0;
     end
//store
6'b010101 :
  begin
        i0we=1'b0;
        a0we=1'b0;
        a1we=1'b0;
        changestate=1'b1;
     end
//add
6'b011010 :
  begin
        i0we=1'b0;
     end
//mova
6'b011110 :
  begin
        i0we=1'b0;
        a_sel=2'b10;
```

```
       end
  //shl1
  6'b100000 :
    begin
       i0we=1'b0;
       a_sel=2'b01;
    end
//shl2
  6'b100010 :
    begin
       i0we=1'b0;
       a_sel=2'b01;
       shift_sel=1'b1;
    end
  endcase
end
endmodule
```

B.2 - REG3 Verilog Code -

```
module reg3(ir_in, ir_addrout, data_in,
                              dataaddr,clock,reset,data_out,data_we);

input clock,reset;
input [15:0] data_in;
input [15:0] ir_in;
output [15:0] dataaddr;
output [15:0] ir_addrout;
output [15:0] data_out;
output data_we;

wire [15:0] ir_out;
wire [15:0] r1_out, r2_out, regin, alu_out, shift_out, b_in;
wire [1:0] alub_sel, regin_sel;
wire [2:0] pcload_sel;
wire regw,rw,pcwe,changestate,addrout_sel;
wire shift_sel;
reg state;

regfile u1sub1 (r1_out,r2_out,ir_out[10:3],regin,clock,regw);
addrout_mux u1sub2 (r2_out,ir_out[5:0], dataaddr, addrout_sel);
ir u1sub3 (clock, ir_in,ir_out);
pcload_mux u1sub4 (pcload_sel, pcload_out,r1_out);
pc u1sub5(reset, clock, pcwe,loadpc,(16'h00 | ir_out[5:0]),ir_addrout);
regin_mux u1sub6(data_in, alu_out,shift_out,r2_out,regin_sel,regin);
alub_mux u1sub7(r2_out,alub_sel,b_in);
alu u1sub8(r1_out,b_in,alu_out);
shifter u1sub9(r1_out,shift_out,shift_sel);
controller u1sub10(ir_out, state, alub_sel, regin_sel, pcload_sel,
       regw,rw,pcwe,changestate,addrout_sel,shift_sel);

assign data_we = rw;

always @(posedge clock)
       begin
              if (changestate)
                     state=~state;
              else
                     state=state;
       end
assign data_out=r1_out;

endmodule

module regfile(r1_out,r2_out,ir,regin,clock,regw);
```

```
input regw,clock;
input [7:0] ir;
input [15:0] regin;
output [15:0] r1_out,r2_out;

dualram #(16,16,5) register_file (ir[7:4],regin,regw,ir[3:0],regin,0,
        r1_out,r2_out);


endmodule


module addrout_mux (r2_out,ir, addrout, addrout_sel);

input [15:0] r2_out;
input [5:0] ir;
input  addrout_sel;
output [15:0] addrout;

assign addrout = (addrout_sel) ? (16'h0000 | ir) : (r2_out);

endmodule


//instruction register
module ir(clock, ir_in,ir_out);

input clock;
input [15:0] ir_in;
output [15:0] ir_out;

reg [15:0] ir_out;
always @(posedge clock)
        begin
                ir_out=ir_in;
        end
endmodule

module pcload_mux(pcload_sel, pcload_out,r1_out);

input [15:0] r1_out;
input [2:0] pcload_sel;

output pcload_out;

assign pcload_out = ((pcload_sel == 3'b100) ? (!r1_out):
                      ((pcload_sel == 3'b011) ? (r1_out):
                      ((pcload_sel == 3'b010) ? (r1_out[15]):
                      ((pcload_sel == 3'b001) ? 1'b1:
                             (1'b0)))));

endmodule

// program counter logic
module pc(reset, clock, pcwe,loadpc,pc_in,pc_out);

input reset,clock,pcwe,loadpc;
input [15:0] pc_in;
output [15:0] pc_out;

reg [15:0] pc_out;

always @(posedge clock)
  begin
    if (reset)
        pc_out=16'h0000;
    else if (pcwe)
```

```
        if (loadpc)
                pc_out= pc_in;
        else
                pc_out=pc_out+1;
  end
endmodule

module regin_mux(data_in, alu_out,shift_out,r2_out,regin_sel,regin);

input [15:0] data_in, alu_out,shift_out,r2_out;
input [1:0] regin_sel;

output [15:0] regin;

assign regin = ((regin_sel== 2'b01) ? r2_out:
                        ((regin_sel == 2'b10) ? data_in:
                        ((regin_sel == 2'b11) ? shift_out:
                          (alu_out))));
endmodule

module alub_mux(r2_out,alub_sel,alub_out);

input [15:0] r2_out;
input [1:0] alub_sel;

output [15:0] alub_out;


assign alub_out = ((alub_sel == 2'b01) ? (~r2_out+1'b1):
                        ((alub_sel == 2'b10) ? 1'b1:
                        (r2_out)));
endmodule

module alu(a_in,b_in,alu_out);

input [15:0] a_in,b_in;
output [15:0] alu_out;

assign alu_out=a_in+b_in;

endmodule

module shifter (r1_out,shift_out,shift_sel);

input [15:0] r1_out;
output [15:0] shift_out;
input shift_sel;

assign shift_out = (shift_sel) ? r1_out<<2 : r1_out<<1;
endmodule


module controller (instruction, state, alub_sel, regin_sel, pcload_sel,
        regw,rw,pcwe,changestate,addrout_sel,shift_sel);

input [15:0] instruction;
input state;
output [1:0] alub_sel, regin_sel;
output [2:0] pcload_sel;
output regw,rw,pcwe,changestate,addrout_sel;
output shift_sel;

reg [1:0] alub_sel, regin_sel;
reg [2:0] pcload_sel;
reg regw,rw,pcwe,changestate,addrout_sel;
reg shift_sel;
```

```
always @(instruction or state) begin

  alub_sel=2'b00;
  addrout_sel=1'b0;
  regin_sel=2'b00;
  regw=1'b1;
  pcload_sel=3'b000;
  rw=1'b0;
  pcwe=1'b1;
  changestate=1'b0;
  shift_sel=1'b0;

casex ({instruction[15:11],state})
  // load
  6'b000000 :
    begin
       regin_sel=2'b10;
       addrout_sel=1'b1;
       changestate=1'b1;
       pcwe=1'b0;
    end
  // load
  6'b000001 :
        begin
       regin_sel=2'b10;
       addrout_sel=1'b1;
       changestate=1'b1;
    end
  // loadr
  6'b000100 :
    begin
       regin_sel=2'b10;
           pcwe=1'b0;
           changestate=1'b1;
    end
  // loadr
  6'b000101 :
    begin
       regin_sel=2'b10;
       changestate=1'b1;
    end
  //storer
  6'b000110 :
    begin
       regw=1'b0;
       rw=1'b1;
       pcwe=1'b0;
       changestate=1'b1;
        end
  //storer
  6'b000111 :
    begin
       regw=1'b0;
       rw=1'b1;
           changestate=1'b1;
        end
  //mov
  6'b000010 :
    begin
       regin_sel=2'b01;
    end
  //add
  6'b001000 :
    begin
    end
  //sub
  6'b001010 :
```

```
    begin
       alub_sel=2'b01;
    end
 //jz
  6'b001100 :
    begin
       regw=1'b0;
       pcload_sel=3'b100;
        end
  //jnz
  6'b001110 :
    begin
       regw=1'b0;
       pcload_sel=3'b011;
        end
  //jneg
  6'b010000 :
    begin
       regw=1'b0;
       pcload_sel=3'b010;
        end
  //goto
  6'b010010 :
    begin
       regw=1'b0;
       pcload_sel=3'b001;
        end
  //inc
  6'b010100 :
    begin
       alub_sel=2'b10;
        end
  //shl1
  6'b010110 :
    begin
       regin_sel=2'b11;
        end
  //shl2
  6'b011000 :
    begin
       regin_sel=2'b11;
       shift_sel=1'b1;
    end
  endcase
end
endmodule
```

Appendix C - Design's ISAs

C.1 - Accumulator's ISAs -

AC1	AC2	AC3	AC4	AC5
loadi I,ADDR	loadi I,ADDR	loadi I,ADDR	loadi ADDR	loadi I,ADDR
loadbyi I,A	loadbyi I,A	loadbyi I,A	loadbyi	loadbyi I,A
storebyi I,A	storebyi I,A	storebyi I,A	storebyi	storebyi I,A
load A,ADDR	load A,ADDR	load A,ADDR	load ADDR	load A,ADDR
store A,ADDR	store A,ADDR	store A,ADDR	store ADDR	store A,ADDR
mova A	mova A	mova A	inci	mova A
inci I	inci I	inci I	deci	inci I
deci I	deci I	deci I	addi	deci I
addi I,A	addi I,A	addi I,A	add ADDR	addi I,A
add A,A	add A,A	add A,A	sub ADDR	add A,A
sub A,A	sub A,A	sub A,A	shl1	sub A,A
shl1 A	shl1 A	shl1 A	jnz ADDR	shl1 A
jnz A,ADDR	jnz A,ADDR	jnz A,ADDR	jz ADDR	jnz A,ADDR
jz A,ADDR	jz A,ADDR	jz A,ADDR	jneg ADDR	jz A,ADDR
jneg A,ADDR	jneg A,ADDR	jneg A,ADDR	goto ADDR	jneg A,ADDR
goto ADDR	goto ADDR	goto ADDR	shl2	goto ADDR
	shl2 A	shl2 A		shl2 A
		je ADDR		je ADDR
		jlt A,ADDR		jlt A,ADDR
		jne ADDR		jne ADDR

C.2 - Register's ISAs -

REG1	REG2	REG3	REG4	REG5
load R,ADDR	load R,ADDR	load R,ADDR	load R,ADDR	load R,ADDR
mov R,R	mov R,R	mov R,R	mov R,R	mov R,R
loadr R,R	loadr R,R	loadr R,R	loadr R,R	loadr R,R
add R,R	add R,R	add R,R	add R,R	add R,R
sub R,R	sub R,R	sub R,R	sub R,R	sub R,R
jz R,ADDR	jz R,ADDR	jz R,ADDR	jz R,ADDR	jz R,ADDR
jnz R,ADDR	jnz R,ADDR	jnz R,ADDR	jnz R,ADDR	jnz R,ADDR
jneg R,ADDR	jneg R,ADDR	jneg R,ADDR	jneg R,ADDR	jneg R,ADDR
goto ADDR	goto ADDR	goto ADDR	goto ADDR	goto ADDR
inc R	inc R	inc R	inc R	inc R
shl1 R,R	shl1 R,R	shl1 R,R	shl1 R,R	shl1 R,R
		shl2 R,R	shl2 R,R	shl2 R,R
			storer R,R	