APPENDIX 1

MIDACC ARCHITECTURE

A1.1 INTRODUCTION

Figure. A1.1 shows the functional block diagram of MIDACC indicating different functional modules other than MIDACC extender.

Figure. A1.1: Functional block diagram of MIDACC

A1.2 MIDACC INTERNALS

MIDACC’s code analyser is named MIDA and the code converter, MICC. The MIDACC extender is an independent software tool for estimating the scope for compound and composite instructions for HIE-MIPS.
A1.2.1 MIDA Internals

Given a MIPS32 code, MIDA profiles the code and produces the following statistics.

A1.2.1.1 Instruction class distribution

The following are the steps involved in finding the instruction class distribution:

1) Scan the instruction
2) Identify the instruction and the type using table A3.1 and maintain the count for each instruction type
3) Increment the instruction address by 4 and repeat steps 1-3 for all the instructions.
4) Add the count of instructions belonging to the same type to get the total count for each instruction type.

A1.2.1.2 Instruction distribution

The following are the steps involved in instruction distribution identification:

1) Scan the instruction
2) Using table A3.1 identify the instruction name. Maintain a count for each instruction.
3) Increment the instruction address by 4 and repeat steps (1) and (2) for all the instructions.
**A1.2.1.3 MIPS Code redundant 0’s Distribution**

The following steps are used to find the MIPS code redundant 0’s distribution:

1) Scan the instruction
2) Identify the instruction as per A1.2.1.2. Maintain a count for each instruction. Identify the RZ value of the instruction using table 4.4.
3) Increment the instruction address by 4 and repeat steps (1) and (2) until reaching the end of the program.
4) Multiply the count of each instruction by corresponding RZ value and add the values obtained for all instructions to get the TRZ value.

**A1.2.1.4 Branch instruction distribution**

The following steps are used to find the branch instruction distribution:

1) Scan the instruction
2) Identify the type of each instruction using table A3.1. If instruction type is branch, then go to step 3. Otherwise go to step 4.
3) Identify the instruction and maintain a count for each branch instruction.
4) Increment the instruction address by 4 and repeat steps (1) and (2) for all the instructions.
A1.2.1.5 WASTIO Calculation

The algorithm in Figure. A1.2 is used to compute the WASTIO percentage

<table>
<thead>
<tr>
<th>Input</th>
<th>Object code of program</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>WASTIO Percentage</td>
</tr>
</tbody>
</table>

Algorithm

```
for each instruction in the program
   if instruction format == I || instruction format == O then
      Read the value immediate/offset field of the instruction
      if all zeros in both LSB and MSB then
         a=a+1
      else if all zeros in LSB then
         b=b+1
      else if all zeros in MSB then
         c=c+1
      else
         do nothing
      fi
   fi
end
WASTIO=2a+b+c
WASTIO Percentage = (WASTIO/Object code size) * 100
```

Figure. A1.2: Algorithm for WASTIO calculation

A1.2.1.6 Population of FTFI

The algorithm in Figure. A1.3 is used to compute the population of FTFI.
Input: Object code of program
Output: Population of FTFI

Algorithm:
1. addu_count=0
2. addiu_count=0
3. lw_count=0
4. sw_count=0
5. for each instruction in the program
   6. Read the value OP and OPX field of the instruction
   7. if OP==000000 && OPX=10000 then
      8.     addu_count=addu_count+1
   9. else if OP==001001 then
      10.    addiu_count=addiu_count+1
   11. else
   12.     do nothing
   13. fi
   14. Check the hex value of MSB of the instruction
   15. if value==0x8C || value==0x8D || value==0x8E || value==0x8F then
   16.     lw_count=lw_count+1
   17. else if value==0xAC || value==0xAD || value==0xAE || value==0xAF then
   18.     sw_count=sw_count+1
   19. else
   20.     do nothing
   21. fi
22. end

FTFI=addu_count+addiu_count+lw_count+sw_count

Figure. A1.3: Algorithm for Population of FTFI

A1.2.1.7 Registers usage behaviour

The following are the steps involved in finding the registers usage behaviour:

1) Scan the instruction
2) Identify the instruction type and format using Table A3.1
3) If instruction type is ALU and format is R, then go to step 4. If
the instruction type is ALU and format is I, then go to step 5.
Otherwise go to step 6.
4) For R-format instructions, check the MSB of rs, rt and rd fields
for each combination and maintain a count of each instruction
for each pattern.
5) For I-format instructions, check the MSB of rs and rt fields for
each combination and maintain a count of each instruction for
each pattern.
6) Increment the instruction address by 4 and repeat steps (1) and
(2) for all the instructions.

A1.2.1.8 Shift length usage

The algorithm in Figure. A1.4 is used to compute the shift length
usage.

<table>
<thead>
<tr>
<th>Input</th>
<th>Object code of program</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>Percentage of shift amount between 16-31</td>
</tr>
<tr>
<td>Algorithm</td>
<td></td>
</tr>
<tr>
<td>1 for each instruction in the program</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Read the value of OP and OPX field</td>
</tr>
<tr>
<td>3 if OP==000000 &amp;&amp; OPX==000000 then</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Read the value of MSB of sa field</td>
</tr>
<tr>
<td>5 if MSB==0 then</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>sll_count_zero=sll_count_zero+1</td>
</tr>
<tr>
<td>7 else</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>sll_count_one=sll_count_one+1</td>
</tr>
<tr>
<td>9 fi</td>
<td></td>
</tr>
<tr>
<td>10 else if OP==000000 &amp;&amp; OPX=000011 then</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Read the value of MSB of sa field</td>
</tr>
<tr>
<td>12 if MSB==0 then</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>sra_count_zero=sra_count_zero+1</td>
</tr>
<tr>
<td>14 else</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>sra_count_one=sra_count_one+1</td>
</tr>
</tbody>
</table>
else if \( \text{OP}==000000 \) && \( \text{OPX}=000010 \) then 

Read the value of MSB of sa field

if MSB==0 then

\[ \text{srl\_count\_zero} = \text{srl\_count\_zero} + 1 \]

else

\[ \text{srl\_count\_one} = \text{srl\_count\_one} + 1 \]

fi

else 

do nothing 

fi

doi

end

Total usage of shift amount= \( \text{sll\_count\_zero} + \text{sra\_count\_zero} + \text{srl\_count\_zero} \)

Shift amount between 16 and 31= \( \text{sll\_count\_one} + \text{sll\_count\_one} + \text{sll\_count\_one} \)

Percentage Shift amount between 16 and 31 

=100* ((Shift amount between 16 and 31) / (Total usage of shift amount))

**Figure. A1.4: Algorithm for Shift Length usage computation**

**A1.2.1.9 Immediate field usage pattern**

The following are the steps involved in finding the immediate field usage pattern:

1) Scan the instruction

2) Identify the instruction type and format using Table A3.1

3) If instruction type is ALU and format is R, then go to step 4. Otherwise go to step 5.

4) Check the immediate fields for all combinations and maintain a count of each instruction for each combination.

5) Increment the instruction address by 4 and repeat steps (1) and (2) for all the instructions.
A1.2.1.10 Offset field usage pattern

The following are the steps involved in finding the offset field usage pattern:

1) Scan the instruction
2) Identify the instruction format using Table A3.1
3) If the instruction format belongs to offset, then go to step 4. Otherwise go to step 5.
4) Check the offset fields for all combinations and maintain a count of each instruction for each combination.
5) Increment the instruction address by 4 and repeat steps (1) and (2) for all the instructions.

A1.2.2 MICC Internals

A1.2.2.1 HIE1 code conversion

Figure. A1.5 shows the algorithm for HIE1 code conversion.

<table>
<thead>
<tr>
<th>Input</th>
<th>MIPS32 instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>HIE1 instruction</td>
</tr>
<tr>
<td>Algorithm</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>for all the instructions</td>
</tr>
<tr>
<td>2</td>
<td>Retain opcode of all instruction as such</td>
</tr>
<tr>
<td>3</td>
<td>Identify HIE1 instruction type using table A4.1</td>
</tr>
<tr>
<td>4</td>
<td>if(HIE1 type==H &amp;&amp; HIE1 type==I)</td>
</tr>
<tr>
<td>5</td>
<td>Retain instruction as such</td>
</tr>
<tr>
<td>6</td>
<td>Go to the next instruction</td>
</tr>
<tr>
<td>7</td>
<td>else if(HIE1 type==A)</td>
</tr>
<tr>
<td>8</td>
<td>Delete all other existing fields</td>
</tr>
<tr>
<td>9</td>
<td>Insert 2-bit iid field with value as shown in table A4.1</td>
</tr>
<tr>
<td>10</td>
<td>else if(HIE1 type==B)</td>
</tr>
</tbody>
</table>
Insert 1-bit iid field with value as shown in table A4.1
Remove 1-bit from rt field
Retain rd field as such

else if(HIE1 type==C)
Remove 1-bit from rd/rs field
Retain function bits as such

else if(HIE1 type==D)
Remove 1-bit each from rs, rt and rd field
Retain function bits as such

else if(HIE1 type==E)
Remove 1-bit each from rt, rd and sa field
Retain function bits as such

else if(HIE1 type==F)
Remove 1-bit each from rs and rt field
Delete 6 unused 0’s
Retain function bits as such

else if(HIE1 type==G)
Insert 2-bit hl field using hl HIE1 MIPS encoding in table 4.5

end

Figure. A1.5 : HIE1 code conversion algorithm

A1.2.2.2 HIE2 code conversion

Figure. A1.6 shows the algorithm for HIE2 code conversion.
Input: MIPS32 instruction
Output: HIE2 instruction

Algorithm
1 for all the instructions
   2 Identify HIE2 instruction type using table A5.1
   3 if(HIE2 type==H && HIE2 type==I)
      4 Retain instruction as such
      5 Go to the next instruction
   6 fi
   7 if(HIE2 type==G1 || HIE2 type==G2 || HIE2 type==G3)
      8 insert it field with it=0
      9 fi
   10 if(HIE2 type==G1)
      11 insert 1-bit hl field using hl encoding in table A5.2
      12 else if(HIE2 type==G2)
      13 insert 2-bit hl field using hl encoding in table A5.3
      14 else
      15 insert 2-bit hl field using hl encoding in table A5.4
     fi
   16 else
   17 insert it field with it=1
   18 fi
   19 Replace 6-bit MIPS32 opcode with 5-bit HIE2 opcode using table A5.1
   20 if(HIE2 type==A && HIE2 type==E && HIE2 type==G3)
   21 insert 2-bit iid field with value as shown in table A5.1
   22 fi
   23 if(HIE2 type==C && HIE2 type==D1 && HIE2 type==D2)
   24 insert 3-bit iid field with value as shown in table A5.1
   25 fi
26 end

Figure. A1.6: HIE2 code conversion algorithm

A1.2.2.3 RMA Code Conversion

The Figure. A1.7 depicts the overview of RMA code conversion process for the sequence “load instruction followed by ALU type instruction”. The Figure. A1.8 depicts the overview of RMA code conversion process for the sequence “ALU type instruction followed by Store instruction”.

231
Figure. A1.7: Overview of RMA code conversion for load sequence
Figure. A1.8: Overview of RMA code conversion for store sequence
A1.2.2.4 RMA+HIE1 code conversion

The Figure. A1.9 depicts the overview of RMA+HIE1 code conversion process.

![Diagram of RMA+HIE1 code conversion]

**Figure. A1.9: RMA+HIE1 code conversion**

A1.2.2.5 RMA+HIE2 code conversion

The Figure. A1.10 depicts the overview of RMA+HIE2 code conversion process.

![Diagram of RMA+HIE2 code conversion]

**Figure. A1.10: RMA+HIE2 code conversion**

A1.3 MIDACC EXTENDER

The MIDACC Extender estimates the scope for the following three requirements: use of compound instructions for D1 and E type instructions in HIE2 code, conversion of the *add* and *addu* instructions having same register for both the source operands into two-address instructions, and use of two composite instructions: *loadmultiple* and *storemultiple*. 