Implement a MIPS processor using SME

Carl-Johannes Johnsen

Department of Computer Science
University of Copenhagen

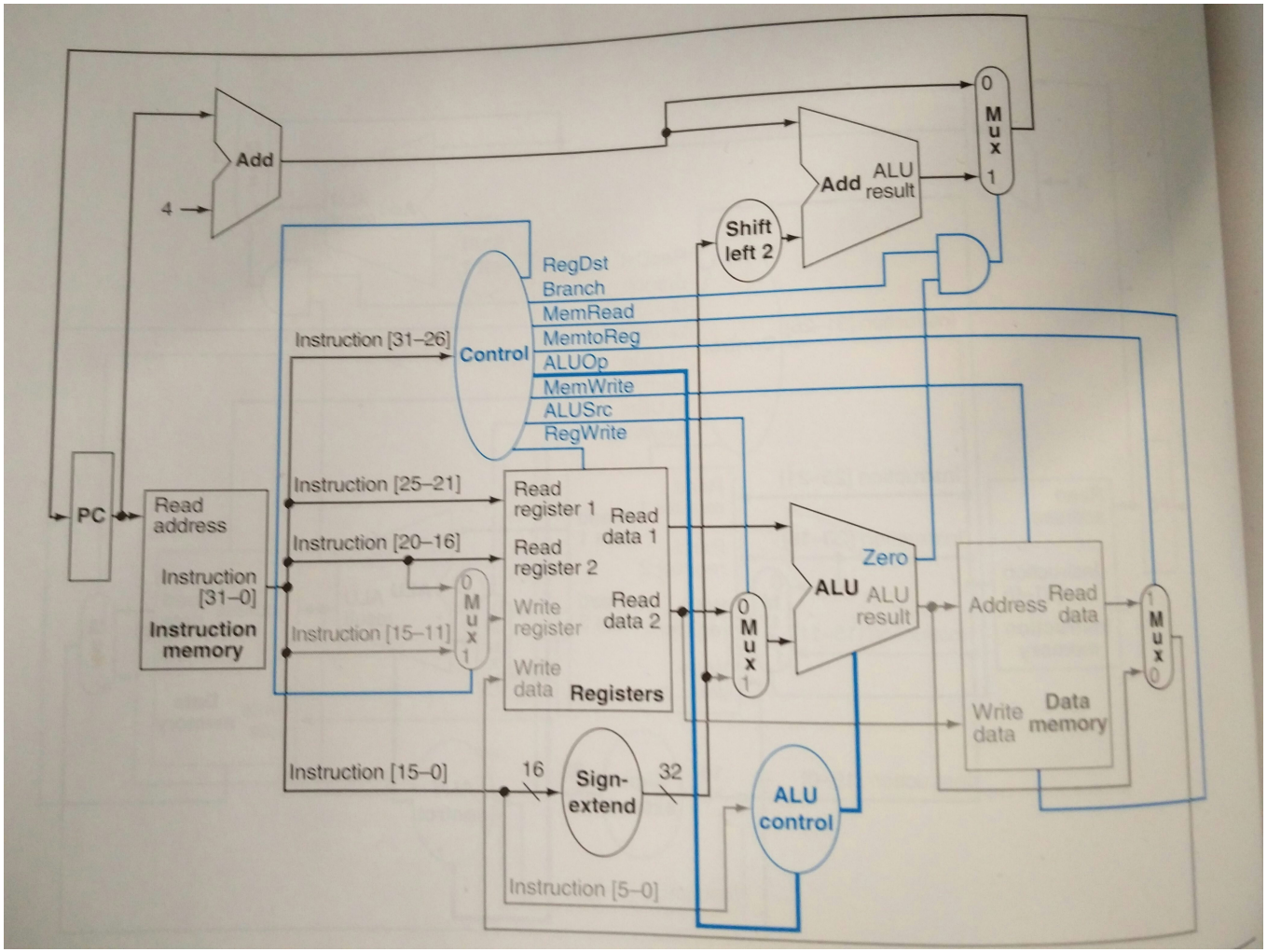
February 14, 2017

Introduction

The goal is to implement a MIPS processor using SME
Will follow the same procedure as the old ARK course on DIKU

Single cycle

The first step is to construct a single cycle MIPS processor, i.e. in one clock cycle, exactly one instruction is executed.



Register file

../sme/src/Examples/SingleCycleMIPS/ID.cs

```
285         public class Register : SimpleProcess
286         {
287             [InputBus]
288             ReadA readA;
289             [InputBus]
290             ReadB readB;
291             [InputBus]
292             WriteEnabled regWrite;
293             [InputBus]
294             WriteAddr writeAddr;
295             [InputBus]
296             WriteData writeData;
297 
298             [OutputBus]
299             OutputA outputA;
300             [OutputBus]
301             OutputB outputB;
302 
303             //int[] data = new int[32];
304             int[] data = Enumerable.Repeat(0, 32).ToArray();
305 
306             protected override void OnTick()
307             {
308                 if (regWrite.flg && writeAddr.val > 0)
309                 {
310                     data[writeAddr.val] = writeData.data;
311                 }
312                 outputA.data = data[readA.addr];
313                 outputB.data = data[readB.addr];
314             }
```

Write buffer

../sme/src/Examples/SingleCycleMIPS/WB.cs

```
26          [ClockedProcess]
27          public class WriteBuffer : SimpleProcess
28          {
29              [InputBus]
30              ID.MuxOutput addrIn;
31              [InputBus]
32              BufIn dataIn;
33              [InputBus]
34              RegWrite regwriteIn;
35
36              [OutputBus]
37              ID.WriteAddr addrOut;
38              [OutputBus]
39              ID.WriteData dataOut;
40              [OutputBus]
41              ID.WriteEnabled regwriteOut;
42
43              protected override void OnTick()
44              {
45                  addrOut.val = addrIn.addr;
46                  dataOut.data = dataIn.data;
47                  regwriteOut.flg = regwriteIn.flg;
48              }
49          }
```

ALU

../sme/src/Examples/SingleCycleMIPS/EX.cs

```
202         protected override void OnTick()
203         {
204             int tmp = -1;
205             switch ((ALUOps) op.val)
206             {
207                 case ALUOps.sr:
208                     tmp = (int) (unchecked((uint) inA.data) >> inB.data);
209                     break;
210                 case ALUOps.sl:
211                     tmp = (int) (unchecked((uint) inA.data) << inB.data);
212                     break;
213                 case ALUOps.sra:
214                     tmp = inA.data << inB.data;
215                     break;
216                 case ALUOps.add:
217                     tmp = inA.data + inB.data;
218                     break;
219                 case ALUOps.addu:
220                     tmp = (int) (((uint)inA.data) + ((uint)inB.data));
```

../sme/src/Examples/SingleCycleMIPS/EX.cs

```
261             result.data = tmp;
262             zero.flg = tmp == 0;
263         }
```

Control Unit

../sme/src/Examples/SingleCycleMIPS/ID.cs

```
239             protected override void OnTick()
240             {
241                 // flag format = [JAL, Jump, RegDst, ALUSrc, MemToReg, RegWrite, MemRead, MemWrite, Branch
242                 short flags = 0; // nop
243                 ALUOpcodes alu = 0; // nop
244                 switch ((Opcodes)input.opcode)
245                 { // The comments are the flags, X is dont care
246                     case Opcodes.Rformat: flags = 0x048;                          break; // 0 0100 1000
247                     case Opcodes.lw:      flags = 0x03B; alu = ALUOpcodes.add; break; // 0 0011 1100
248                     case Opcodes.sw:      flags = 0x022; alu = ALUOpcodes.add; break; // 0 001X 0010
249                     case Opcodes.beq:     flags = 0x001; alu = ALUOpcodes.sub; break; // 0 0X0X 0001
250                     case Opcodes.addi:    flags = 0x028; alu = ALUOpcodes.add; break; // 0 0010 1000
251                     // default: flags = 0; alu = 0; break;
252                 }
253                 jal.flg      = ((flags >> 8) & 1) == 1;
254                 jump.flg     = ((flags >> 7) & 1) == 1;
255                 regdst.flg   = ((flags >> 6) & 1) == 1;
256                 alusrc.flg   = ((flags >> 5) & 1) == 1;
257                 memtoreg.flg = ((flags >> 4) & 1) == 1;
258                 regwrite.flg = ((flags >> 3) & 1) == 1;
259                 memread.flg  = ((flags >> 2) & 1) == 1;
260                 memwrite.flg = ((flags >> 1) & 1) == 1;
261                 branch.flg   = ( flags       & 1) == 1;
262                 aluop.code   = (byte)alu;
263             }
```

ALU control

../sme/src/Examples/SingleCycleMIPS/EX.cs

```
161             protected override void OnTick()
162             {
163                 if (op.code == (byte)ALUOpcodes.RFormat) // R format
164                 {
165                     switch ((Funcs)funct.val)
166                     {
167                         case Funcs.add: output.val = (byte)ALUOps.add; break;
168                         case Funcs.sub: output.val = (byte)ALUOps.sub; break;
169                         case Funcs.and: output.val = (byte)ALUOps.and; break;
170                         case Funcs.or : output.val = (byte)ALUOps.or;  break;
171                         case Funcs.slt: output.val = (byte)ALUOps.slt; break;
172                         default: output.val = 0; break; // nop
173                     }
174                 }
175                 else
176                 {
177                     switch ((ALUOpcodes)op.code)
178                     {
179                         case ALUOpcodes.add: output.val = (byte)ALUOps.add; break;
180                         case ALUOpcodes.sub: output.val = (byte)ALUOps.sub; break;
181                         default: output.val = 0; break; // nop
182                     }
183                 }
184             }
```

Splitter and Sign extend

../sme/src/Examples/SingleCycleMIPS/ID.cs

```
163         protected override void OnTick()
164         {
165             uint tmp = instr.instruction;
166             byte opcode = (byte) ((tmp >> 26) & 0x3F);
167             byte rs =     (byte) ((tmp >> 21) & 0x1F);
168             byte rt =     (byte) ((tmp >> 16) & 0x1F);
169             byte rd =     (byte) ((tmp >> 11) & 0x1F);
170             byte funct =  (byte) ( tmp        & 0x3F);
171 
172             readA.addr = rs;
173             readB.addr = rt;
174             mux.rd = rd;
175             mux.rt = rt;
176             control.opcode = opcode;
177             signExt.data = (short) (tmp & 0xFFFF); // Last 16 bit
178             aluFunct.val = funct;
179         }
```

../sme/src/Examples/SingleCycleMIPS/ID.cs

```
206         protected override void OnTick()
207         {
208             output.data = input.data;
209         }
```

Instruction memory

../sme/src/Examples/SingleCycleMIPS/IF.cs

```
 98          protected override void OnTick()
 99          {
100              int i = addr.address;
101              if (i >= 0 && i < program.Length)
102              {
103                  instr.instruction = program[i];
104                  shut.running = true;
105              }
106              else
107              {
108                  instr.instruction = 0x0; // nop
109                  shut.running = false;
110              }
111          }
```

Test program and output

../sme/src/Examples/SingleCycleMIPS/IF.cs

```
78              uint[] program =
79              {
80                  0x20010005, // addi r1 r0 0x5 - 5
81                  0x20020002, // addi r2 r0 0x2 - 2
82                  0x00221820, // add r3 r1 r2 - 7
83                  0x00232020, // add r4 r1 r3 - 12
84                  0x00842820, // add r5 r4 r4 - 24
85                  0x00A13022, // sub r6 r5 r1 - 19
86                  0x00233824, // and r7 r1 r3 - 5
87                  0x00234025, // or  r8 r1 r3 - 7
88                  0x0023482A, // slt r9 r1 r3 - 1
89                  0x0061502A, // slt r10 r3 r1 - 0
90                  0xAD2B0008, // sw r11 0x8 r9 - 9 -- should not write to register
91                  0x8D2B0008, // lw r11 0x8 r9 - 9
92                  0x10270002, // beq r1 r7 0x2 - 0 -- should not write to register, but should jump the next
93                  0x200C0003, // addi r12 r0 0x3 - 3 -- should not be executed
94                  0x200C0003, // addi r12 r0 0x3 - 3 -- should not be executed
95                  0x200C000F, // addi r12 r0 0xF - 15
96              };
```

After the program has run, the register file has the following contents:

```
0, 5, 2, 7, 12, 24, 19, 5, 7, 1, 0, 9, 15, 0,  0,  0,
0, 0, 0, 0, 0,  0,  0,  0, 0, 0, 0, 0, 0,  0,  0,  0
```

Future work

- Construct the Memory unit
- Add support for Jump instructions
- Add more instructions
- Pipelining