TEXAS STATE®

CS 3339 – Computer Architecture
Project 1 Discussion

Instructor
Mr. Lee Hinkle

With deep acknowledgement to Dr. Martin Burtscher
and Ms. Molly O'Neil

image credit http://www.popgun80sband.co.uk/blog/top-ten-robots-of-the-80s/

Big Picture

There are a total of 6 projects this semester

- Project 1 – Disassembler (machine code to assembly)
- Project 2 – Emulator (machine code to operation)
- Project 3 – Pipelining (simulator for cycle timing)
- Project 4 – Pipelining II (add forwarding paths)
- Project 5 – Caching (add simulated data cache)
- Project 6 – Parallel Processing

The projects are more important than their point values reflect – they reinforce key topics

Input Files .mips

In binary format

```
exeFile.open(argv[1], ios::binary | ios::in);
```

In c++ :: is the scope resolution operator for accessing static variables and methods of a class/struct or namespace
class-name :: identifier
namespace :: identifier

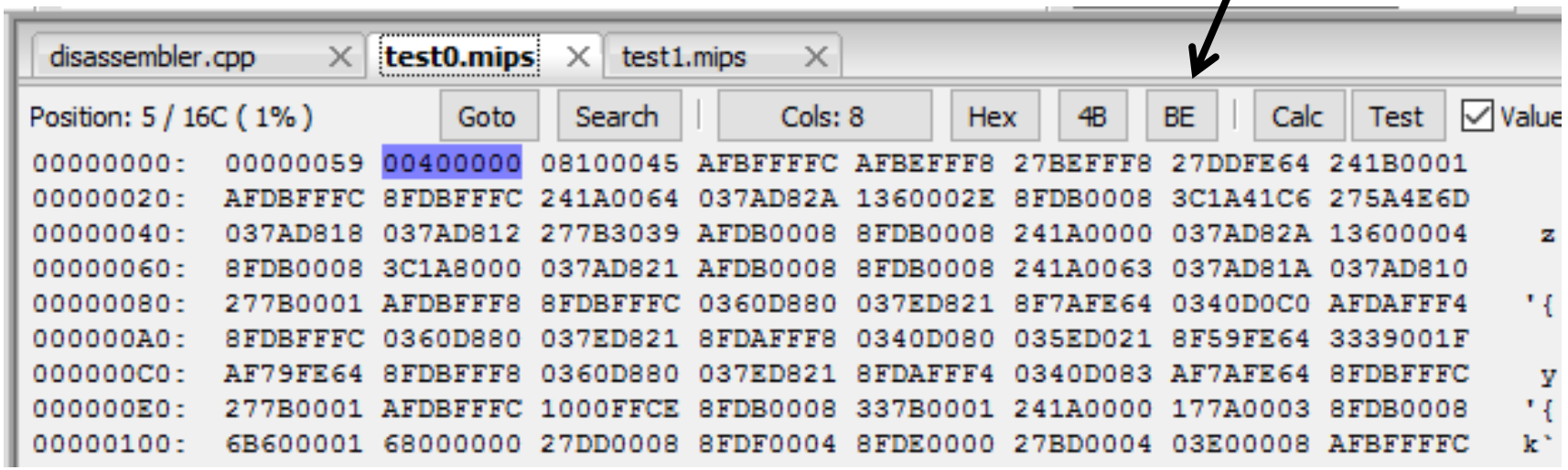
On the MIPS reference card :: is concatenation of bit fields

This is not necessary to complete the project, just a way to look if you are curious.

Viewing in code::blocks

File -> Open with Hex Editor

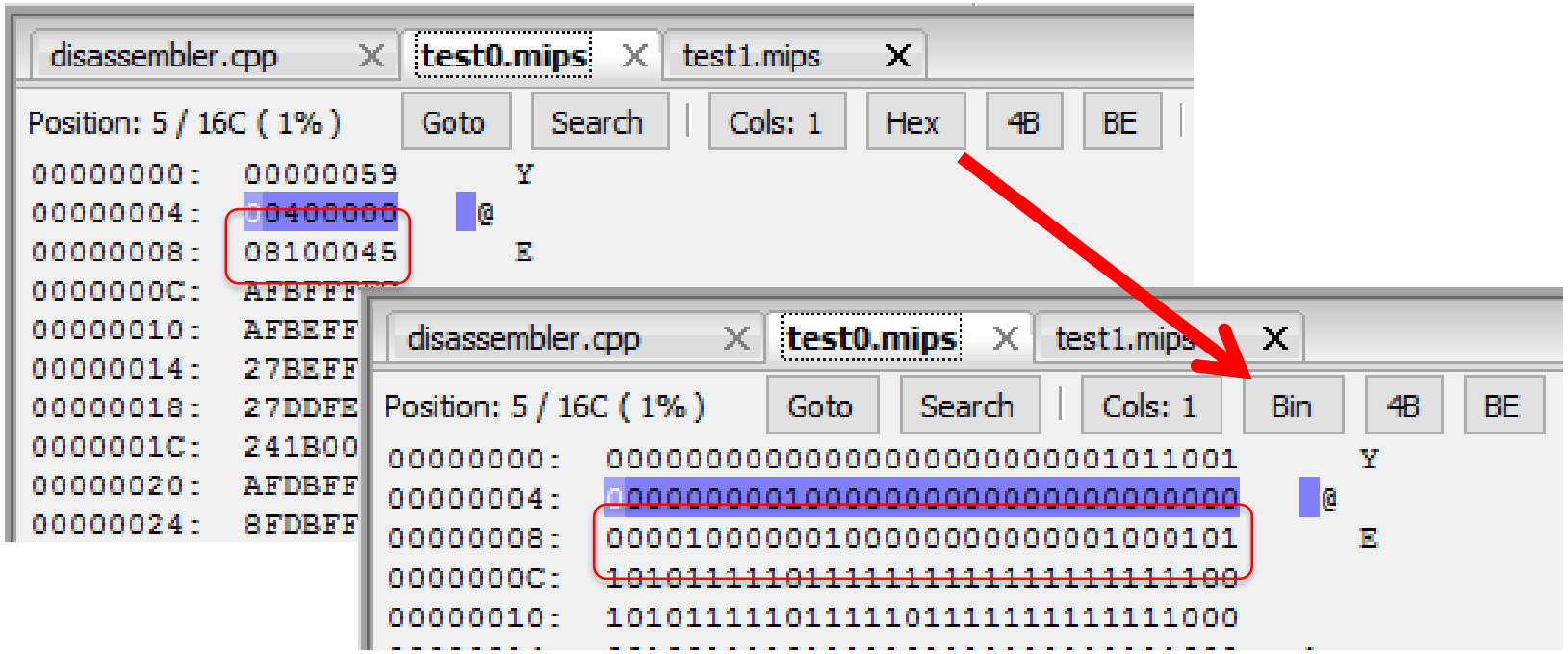
Big Endian or Little Endian



1st word is count 0x59 instructions

2nd word is starting address 0x00400000

Analyze 1st Instruction



0000 1000 0001 0000 0000 0000 0100 0101

1st 6 bits [31:26] are the opcode 000010b
= 02h means jump

Extracting info from larger field

>> is the C++ shift right operation

Example: I want to find value of field ans = B[3:2]

```
   B = 0010 1010 (shift right by 2 places)
 ans = 0000 1010 (leading digits still wrong)
```

& is the C++ bitwise 'and' operation

Very useful to extract bits you care about from larger inputs

```
   ans = 0000 1010
  mask = 0000 0011 (put 1 where you care)
result = 0000 0010 (after 'B & mask;')
```

Combine

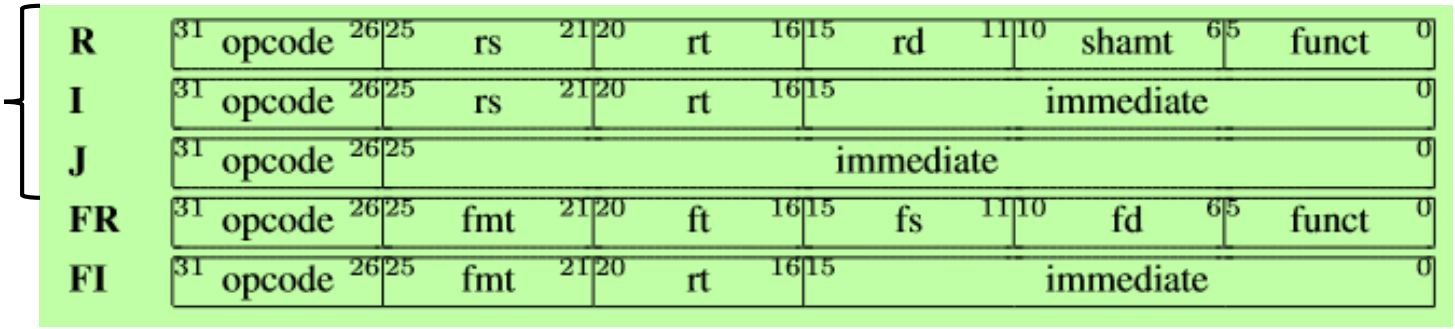
```
ans = (B >> 2) & 0x3;
ans = 0000 0010 or 0x2;
```

Parsing…

```
void disassembleInstr(uint32_t pc, uint32_t instr) {
  uint32_t opcode;      // opcode field
  uint32_t rs, rt, rd;  // register specifiers
  uint32_t shamt;       // shift amount (R-type)
  uint32_t funct;       // funct field (R-type)
  uint32_t uimm;        // unsigned version of immediate (I-type)
  int32_t simm;         // signed version of immediate (I-type)
  uint32_t addr;        // jump address offset field (J-type)

  opcode = /* FIXME */
  rs = /* FIXME */
  rt = /* FIXME */
  rd = /* FIXME */
  shamt = /* FIXME */
  funct = /* FIXME */
  uimm = /* FIXME */
  simm = /* FIXME */
  addr = /* FIXME */
```

We will only use these 3

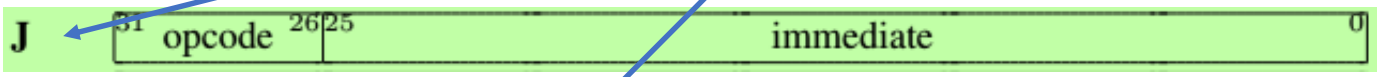


Analyze 1st Instruction

1st 6 bits [31:26] = opcode 02h means jump

0000 1000 0001 0000 0000 0000 0100 0101

Jump	j	J	PC=JumpAddr	(5)	2_{hex}



(3) ZeroExtImm = { 16{1b'0}, immediate }
(4) BranchAddr = { 14{immediate[15]}, immediate, 2'b0 }
(5) JumpAddr = { PC+4[31:28], address, 2'b0 }

PC = 00000000010000000000000000000000

immediate = 00000100000000000001000101

2'b0 = 00

JumpAddr = 00000000010000000000000100010100

0 0 4 0 0 1 1 4

```
CS 3339 MIPS Disassembler
  400000: j 400114
  400004: sw $ra, -4($sp)
```

Watch out for signed versions!

```
uint32_t uimm;        // unsigned version of immediate (I-type)
int32_t simm;         // signed version of immediate (I-type)
```

2's compliment means that when adding leading bits you must use the sign value. Fortunately this is handled by the right shift function

"The right-shift operator causes the bit pattern in shift-expression to be shifted to the right by the number of positions specified by additive-expression. For unsigned numbers, the bit positions that have been vacated by the shift operation are zero-filled. For signed numbers, the sign bit is used to fill the vacated bit positions. In other words, if the number is positive, 0 is used, and if the number is negative, 1 is used."

https://msdn.microsoft.com/en-us/library/336xbhcz.aspx

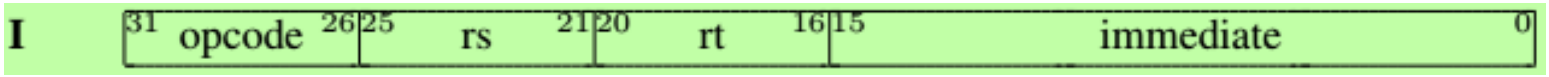
Branch Addressing

```
   PC   :inst(hex): inst assembly (from test0.mips)
4000e0: 1000ffce: beq $zero, $zero, 40001c
```

Branch On Equal	beq	I	if(R[rs]==R[rt]) PC=PC+4+BranchAddr	(4)	4_{hex}

Opcode=4h=beq

0001 0000 0000 0000 1111 1111 1100 1110



Unlike jump branches use relative addresses. Signed numbers are required: could go forward (pos) or backward (neg). When extending a 2's complement signed value you must pad with the sign bit, hence the 14 digits of immediate[15]

if(R[rs]==R[rt]) PC=PC+4+BranchAddr

(4) BranchAddr = {14{immediate[15]},immediate,2'b0 }

1111 1111 1111 1111 1111 1111 0011 1000

14 copies of immediate[15] | immediate | for word resolution (x4)

Branch Addressing (continued)

from previous slide

1111 1111 1111 1111 1111 1111 0011 1000

This is a negative number – recall that to convert you take complement and subtract 1

0000 0000 0000 0000 0000 0000 1100 0111

= -0C7h - 1 = -C8h

PC=PC+4+BranchAddr

PC = 4000e0
Add 4 = 4000e4
Add the negative branch address
4000e4h – C8h = 40001Ch

```
4000dc: sw $k1, -4($fp)
4000e0: beq $zero, $zero, 40001c
4000e4: lw $k1, 8($fp)
```

Note: You should not negate the 2's complement in your code, if it's declared signed (simm) it will work. This slide is just to help you understand the concept.

Excel 2010 English EULA

Snippet – this language is very common in software license agreements.

"7. SCOPE OF LICENSE. The software is licensed, not sold. This agreement only gives you some rights to use the features included in the software edition you licensed. Microsoft reserves all other rights. Unless applicable law gives you more rights despite this limitation, you may use the software only as expressly permitted in this agreement. In doing so, you must comply with any technical limitations in the software that only allow you to use it in certain ways. You may not:

- work around any technical limitations in the software;
- reverse engineer, decompile or disassemble the software, except and only to the extent that applicable law expressly permits, despite this limitation;
- make more copies of the software than specified in this agreement or allowed by applicable law, despite this limitation;"