

CSE 141 Discussion

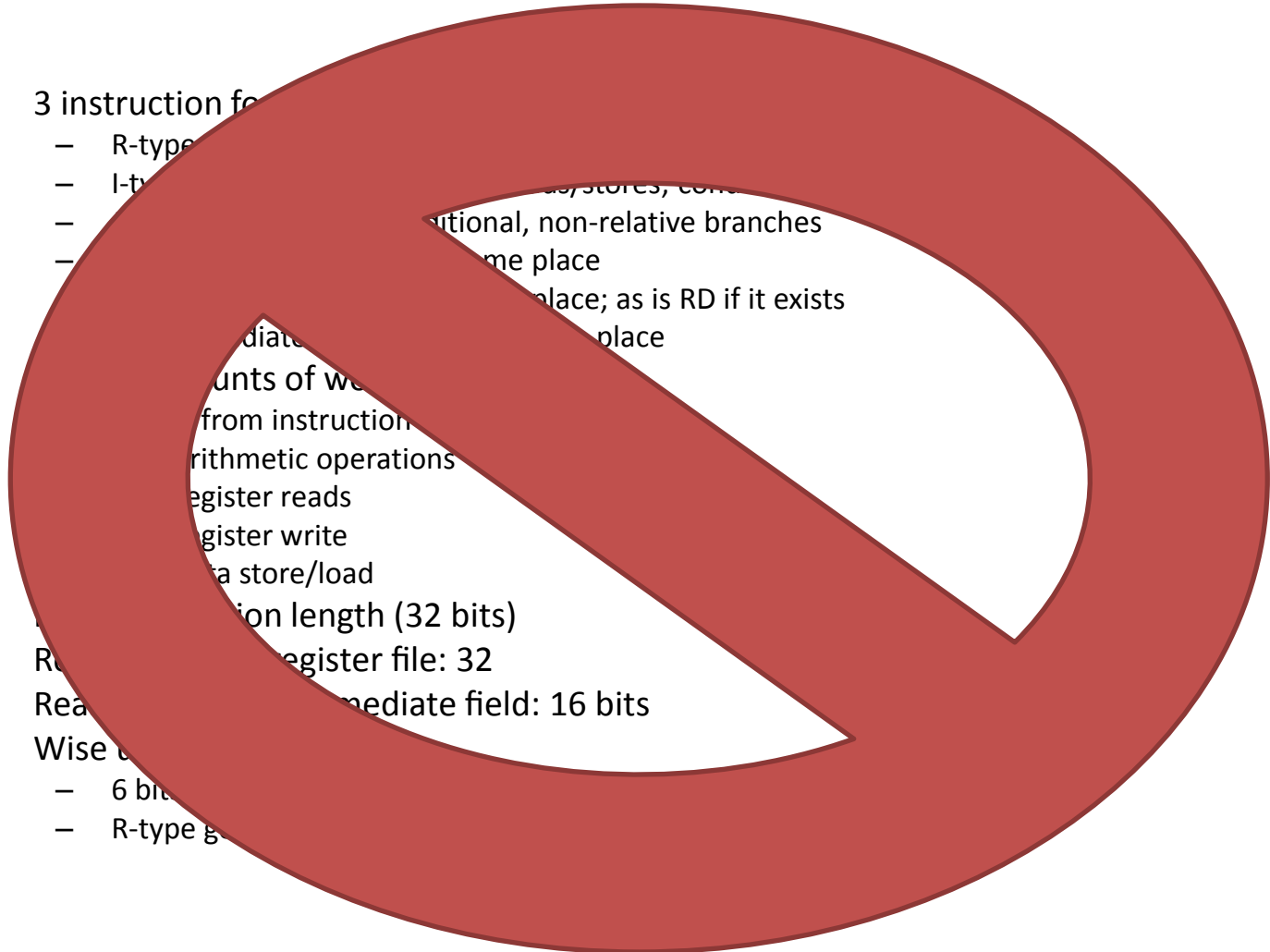
Designing your Project ISA

MIPS ISA

- 3 instruction formats: I, R, and J.
 - R-type: Register-register Arithmetic
 - I-type: immediate arithmetic; loads/stores; cond branches
 - J-type: Jumps - Non-conditional, non-relative branches
 - opcodes are always in the same place
 - rs and rt are always in the same place; as is RD if it exists
 - The immediate is always in the same place
- Similar amounts of work per instruction
 - 1 read from instruction memory
 - ≤ 1 arithmetic operations
 - ≤ 2 register reads
 - ≤ 1 register write
 - ≤ 1 data store/load
- Fixed instruction length (32 bits)
- Relatively large register file: 32
- Reasonably large immediate field: 16 bits
- Wise use of opcode space
 - 6 bits of opcode
 - R-type gets another 6 bits of “function”

3 instruction fo

- R-type
- I-type
- Conditional, non-relative branches
- Same place
- Same place; as is RD if it exists
- Immediate
- Same place
- Units of wo
- from instruction
- arithmetic operations
- register reads
- register write
- store/load
- instruction length (32 bits)
- Register file: 32
- Immediate field: 16 bits
- Wise
- 6 bit
- R-type g

- 
- 3 instruction fo
- R-type
 - I-type
 - Conditional, non-relative branches
 - Same place
 - Same place; as is RD if it exists
 - Immediate
 - Same place
 - Units of wo
 - from instruction
 - arithmetic operations
 - register reads
 - register write
 - store/load
 - instruction length (32 bits)
 - Register file: 32
 - Immediate field: 16 bits
 - Wise
 - 6 bit
 - R-type g

Your ISA

- Several Constraints that MIPS doesn't have:
 - Instruction Width: Choice of two sizes (see website)
 - Data and address widths: 34 bits
 - General purpose enough to run provided benchmarks and (maybe) more general programs
- Which means, above all else, you can't just copy MIPS and call it a day.

Your ISA

- MUST Include the following instructions:
 - *in* [dest] [channel[3:0]] : read a 34-bit data from the specified channel and save at [dest]
 - *out* [src] [channel[3:0]] : write a 34-bit data from [src] to the specified channel
 - *halt* : stop execution and return control to the simulator's command prompt.
 - *la* [reg] [label] : load the address of label into reg; this is actually a pseudo instruction. One way to implement it is to expand it out to a series of *sloi* (shift left and or immediate) instructions that take a value in a register, shift it left by the width of your immediate and then or the immediate. Doing this repeatedly allows you to construct a 34 bit value fairly quickly. No matter how you implement *la*, make sure you can handle a label that has a 34 bit address.

Aside: Pseudo Instructions

- Pseudo Instructions are the assembly version of syntactic sugar
- Made to make coding/designing a program easier
- Examples:
 - move \$rt, \$rs (means $R[rt] = R[rs]$)
 - Equivalent to addi \$rt, \$rs, 0 ($R[rt] = R[rs] + 0$)
 - bgt \$rs, \$rt, Label (means if($R[rs] > R[rt]$), PC=Label)
 - Equivalent to (slt \$al, \$rt, \$rs; bne \$at, \$zero, Label)
 - (slt is “set on less than”, returns 1 if \$rs is less than \$rt)

Your ISA

- So what does it have to implement?
 - Two benchmarks, fibonacci and SuperGarbage

SuperGarbage

```
// function supergarbage:
//   perform various operations
//
// opcodes
// 0: subtract
// 1: right shift
// 2: nor
// 3: swap
// 4: in
// 5: out
// 6: conditional jump with link
// 7: halt
//
// note: int is 34 bits
```

```
struct inst {
    int op;
    int srcA;
    int srcB;
    int dest;
};
```

```
int SuperGarbage(int pc, int *mem)
{
    while(1)
    {
        struct inst *instruction = &(mem[pc]);
        int op = instruction->op;
        int srcA = instruction->srcA;
        int srcB = instruction->srcB;
        int dest = instruction->dest;
        pc = pc + 4;

        switch(op) {
            case 0: mem[dest] = mem[srcA] - mem[srcB]; break;
            case 1: mem[dest] = mem[srcA] >> 1; break;
            case 2: mem[dest] = ~(mem[srcA] | mem[srcB]); break;
            case 3: temp = mem[srcB]; mem[dest] = mem[mem[srcA]]; mem[mem[srcA]] = temp; break;
            case 4: in mem[dest], mem[srcA]; break; // in mem, channel #
            case 5: out mem[srcA], mem[srcB]; break; // out data, channel#
            case 6:
                mem[dest] = pc;
                if (mem[srcA] < 0)
                {
                    pc = mem[srcB];
                }
                break;
            case 7: return pc;
        }
    }
}
```

SuperGarbage

- SuperGarbage is a very simple virtual machine.
- Your ISA needs to be able to do the things that SuperGarbage requires, but it doesn't need to be EXACTLY those instructions

Fibonacci

```
// Recursive Fibonacci  
// get 'n'th fibonacci number .. sort of!  
// You should not alter the algorithm
```

```
int fib(int n)  
{  
    if (n < 0)  
        return 0x3DEADBEEF;  
    else if (n <= 2)  
        return 1;  
    else if (n == 29)  
        return 514229;  
    else if (n == 30)  
        return 832030; // note: this "bug" is intentional and matches with SG fib  
    else if (n == 48)  
        return 4807526976;  
    else if (n == 49)  
        return 7778742049;  
    else return fib(n-1) + fib(n-2);  
}
```

Fibonacci

- Fairly straight forward fibonacci algorithm
- Try to work out necessary instructions for your ISA *after* figuring it out for SuperGarbage.
 - Why? Because its “easier”.
 - And because for the first assignment, you only have to worry about SuperGarbage

Your ISA

- Three Goals
 - Energy-Delay
 - $i * i * (5+W)$, where 'i' is the number of dynamic instructions and 'W' is the instruction width
 - General Purpose
 - If the professor gave you a (slightly) different program, could your ISA do it?
 - Novelty
 - Again, don't copy MIPS

Design Choices

- Instruction Set Width
 - Choice of two different widths
 - Tradeoff of flexibility vs. (possible) energy savings
 - There is no one case that is better than the others.
 - That said, once you pick a side, stick with it.
 - As in, don't have some instructions be one size and others a different size
 - Should definitely have which path you want locked in by the first progress report.

Design Choices

- What do you need? I mean, really, what do you *need*?
 - As you go through the two programs, look for instructions that are absolutely critical.
 - Then look for instructions that will make your life a lot easier, but aren't necessary (if any).
 - The second batch are what you may have to trim/rework later on.

Design Choices

- Opcode Length
 - Too few, and you don't have very many instructions
 - Too many, and you don't have room for immediates or registers
 - Best to find a balance, but what the balance is depends on implementation
 - Also, look into function codes to extend your number of instructions

Design Choices

- Registers
 - Can technically have as many registers as you want, but you are limited by your ISA
 - Try to have some number of general purpose registers
 - But, specialized registers have their place (see the stack pointer in MIPS)
 - Not only that but they can extend your instruction set by letting you do other things with those bits.

Design Choices

- Novelty
 - Again, you aren't writing MIPS, so you shouldn't pretend you are writing a subset of it.
 - Think of instructions that *aren't* in the MIPS language that would be nice to have for your processor.
 - That said, you should justify their existence, because *we* don't know why you added this.
 - Try not to make them, as the writeup says, "brittle".

Design Choices

- Putting it all together:
 - Example: Instruction foo
 - Opcode: 0010 (4-bits)
 - srcA: 000 (3-bits)
 - srcB: 000 (3-bits)
 - Immediate: 0000 (4-bits)
 - How could I make this work for 11 bits?

Design Choices

- Putting it all together
 - I don't have enough space for an add function and a sub function, but I already have something that inverts values from positive to negative.
 - Well...I don't really need the subtraction function anymore do I?
 - For a subtraction, just invert the second register's numbers and add.

Design Choices

- Issues you may run into
 - Too few bits!
 - Yea, that's gonna happen.
 - Try to think of ways to work around it
 - This isn't MIPS!
 - Too many instructions
 - Try to combine like instructions taking care to not sacrifice generality.
 - Instructions aren't general purpose
 - Try not to do this, make sure they are just "instructions to make SG run fast"
 - We are, at the very least, trying to get you from writing a "fib" ISA instruction that does everything...really slowly.