Machine Instruction Encoding

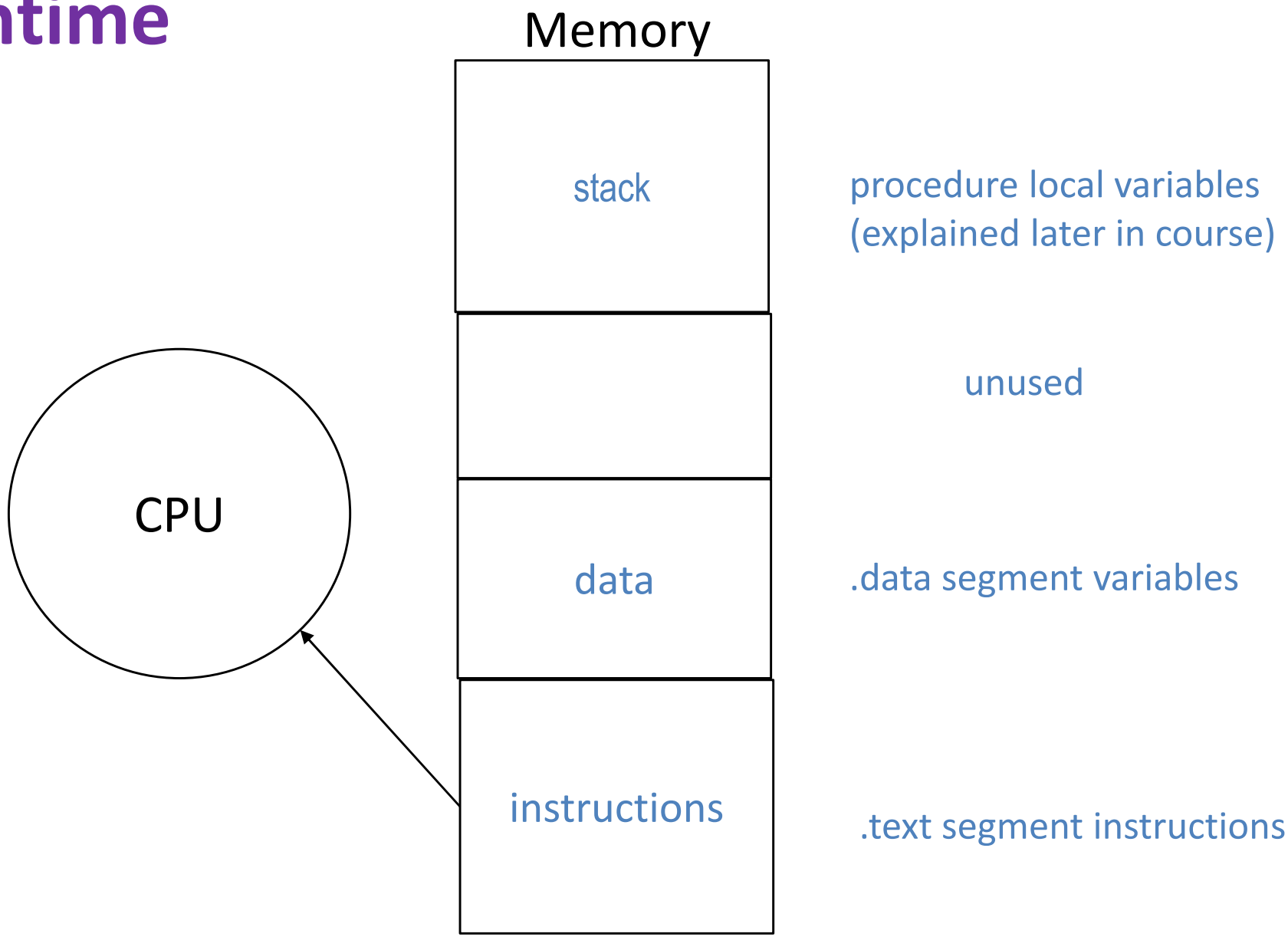
CSE 410 22wi

Lecture 04

Assembly Language / Assemblers

- We have been writing instructions in assembly code
- The goal of assembly code is to be convenient for humans to read/write
- What's convenient for the hardware to read/execute is bit strings
- The main job of the assembler is to convert the string (assembler) representation of instructions into the binary representation
- When the program is run, it is the binary representation that is loaded into memory

Runtime


Memory
stack
procedure local variables
(explained later in course)
unused
CPU
data
.data segment variables
instructions
.text segment instructions

Instruction Encoding

- Example:
  add x15, x14, x15 → 0x00f707b3
  0000 0000 1111 0111 0000 0111 1011 0011
- Goals:
  - Compact: use fewer bits
    - Less memory has to be fetched to execute the program if each instruction is short
  - Easy to decode
    - (In the subset of RISC-V we use) All instructions are the same length
      - 32 bits
    - To the extent possible, all instructions are represented in very similar ways
  - Expressive
    - Example: addi x3, x2, 10
      - Instructions are only 32-bits, so there has to be some limit on the size of the immediate operand

RISC-V Instruction Encoding

- More examples
  - add x15,x14,x15 0x00f707b3
  - addi x8,x2,48 0x03010413
  - lw x1,44(x2) 0x02c12083
  - beq x14,x15, 0x3c 0x00f70a63

- The machine instruction (32-bit string) must encode registers, immediates, offsets, and the operation
- There are only a few classes of instruction types, with multiple operations in each class
  - Example: *add* and *or* both take three register operands

RISC-V Instruction Encoding Classes

CORE INSTRUCTION FORMATS

	31 27	26 25	24 20	19 15	14 12	11 7	6 0
R	funct7		rs2	rs1	funct3	rd	Opcode
I	imm[11:0]			rs1	funct3	rd	Opcode
S	imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode
SB	imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]					rd	opcode
UJ	imm[20|10:1|11|19:12]					rd	opcode

- The opcode field is in a fixed place, bits 0 through 6
- From the opcode you can tell what the format of the instruction is (R, I, S, etc.)
- The opcode, plus sometimes the funct3 field, plus sometimes the funct7 field, tell you what the specific operation is
  - add → opcode: 0110011 funct3: 000 funct7: 0000000
  - sub → opcode: 0110011 funct3: 000 funct7: 0100000

Instruction classes

CORE INSTRUCTION FORMATS

	31 27	26 25	24 20	19 15	14 12	11 7	6 0
R	funct7		rs2	rs1	funct3	rd	Opcode
I	imm[11:0]			rs1	funct3	rd	Opcode
S	imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode
SB	imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]					rd	opcode
UJ	imm[20|10:1|11|19:12]					rd	opcode

- R type
  - add, sub, sll, slt, sltu, xor, srl, sra, or, and,
- I type
  - lw, lb, addi, slli, slti, sltiu, xori, srli, srai, ori, andi
- SB type
  - beq, bne, blt, bge, bltu, bgeu

Example: R type

CORE INSTRUCTION FORMATS

	31 27	26 25	24 20	19 15	14 12	11 7	6 0
R	funct7		rs2	rs1	funct3	rd	Opcode
I	imm[11:0]			rs1	funct3	rd	Opcode
S	imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode
SB	imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]					rd	opcode
UJ	imm[20|10:1|11|19:12]					rd	opcode

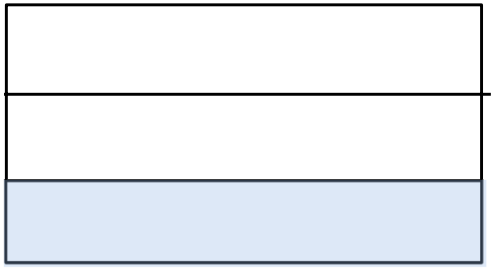
- 0x00f707b3
  - 0000 0000 1111 0111 0000 0111 1011 0011
  - (0000000) (01111) (01110) (000) (01111) (0110011)

    funct7 rs2 rs1 funct3 rd opcode

    x15 x14 x15 add

Actual RISC-V Memory Semantics

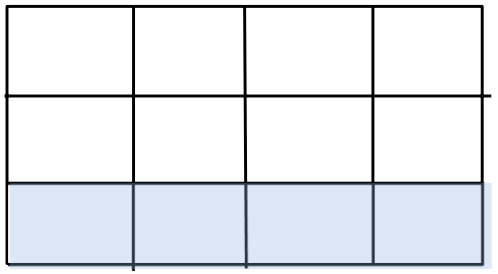
- We've been treating memory as "word addressable"



...

lw x2, 2(x0)

- Memory is actually "byte addressable"
  - A byte is 8 bits



...

lw x2, 8(x0)

Actual Memory Semantics

- word 32 bits lw, sw
  halfword 16 bits lh, sh
  byte 8 bits lb, sb
  double 64 bits ld, sd
- loads and stores must be “aligned”
  - for words, the effective address must be a multiple of 4
  - for halfwords, a multiple of two
  - for bytes, any address is valid
  - for doubleword, a multiple of 8
- Why?
  - simplifies many aspects of implementing the ISA in hardware

Branch Encoding (SB class)

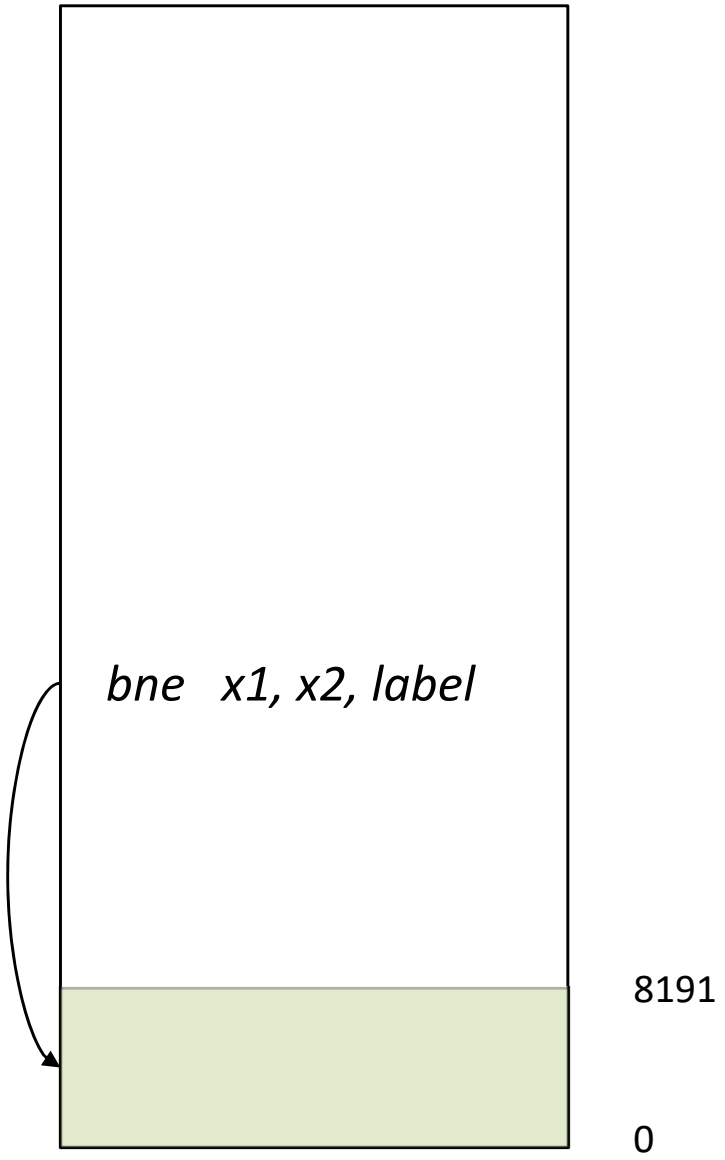
CORE INSTRUCTION FORMATS

	31 27	26 25	24 20	19 15	14 12	11 7	6 0
R	funct7		rs2	rs1	funct3	rd	Opcode
I	imm[11:0]			rs1	funct3	rd	Opcode
S	imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode
SB	imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]					rd	opcode
UJ	imm[20|10:1|11|19:12]					rd	opcode

- beq rs1, rs2, immed
  - immediate is stored in instruction in an odd way
  - high order bit of instruction is sign bit of immediate value
  - bits 30 through 25 are bits 10:5 of immediate, just like I format
  - bits 11 through 8 are bits 4:1 of immediate, just like I format
  - bit 0 isn't stored, because it has to be 0 because of alignment
    - minimum instruction length in all RISC-V versions is 2 bytes
  - there's no where else for bit 11 of the immediate to go than bit 7

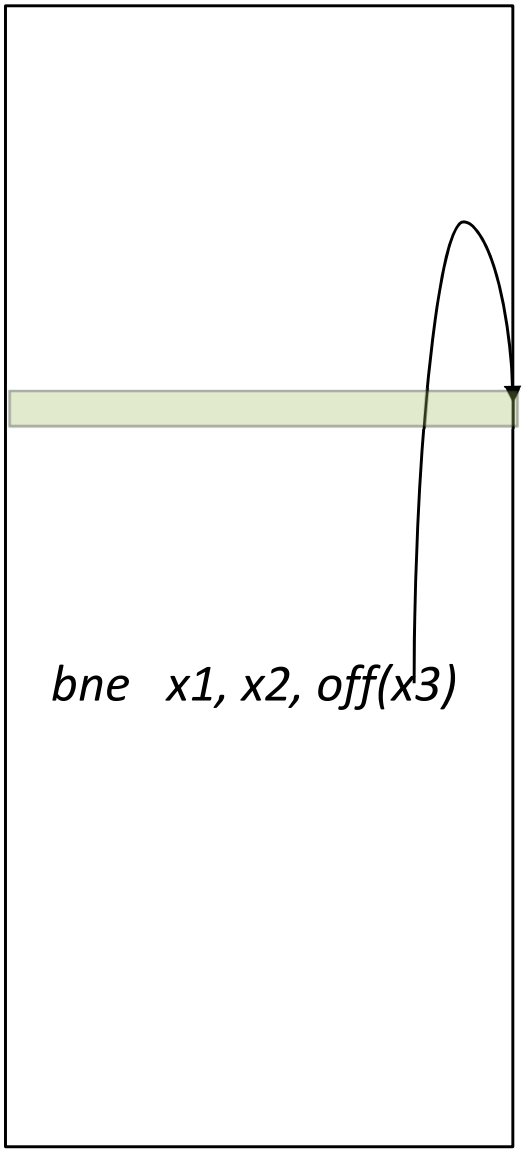
Branch Encoding

- The SB encoding can represent a 13 bit number
- If taken as an unsigned integer, the immediate can range from 0 to 8191
- If the immediate is an absolute address, branch destination is very restricted



Branch Encoding: How to fix this

- Could make the immediate an *offset*
- Offset from what?
  - Specify another register to use as the base address
- Example:
  - bne x1, x2, off(x3)
- Problems:
  1. Need 5 bits to encode base register in instruction
     - offset would be only 8 bits
       - 0 – 255
  2. Have to set the base register

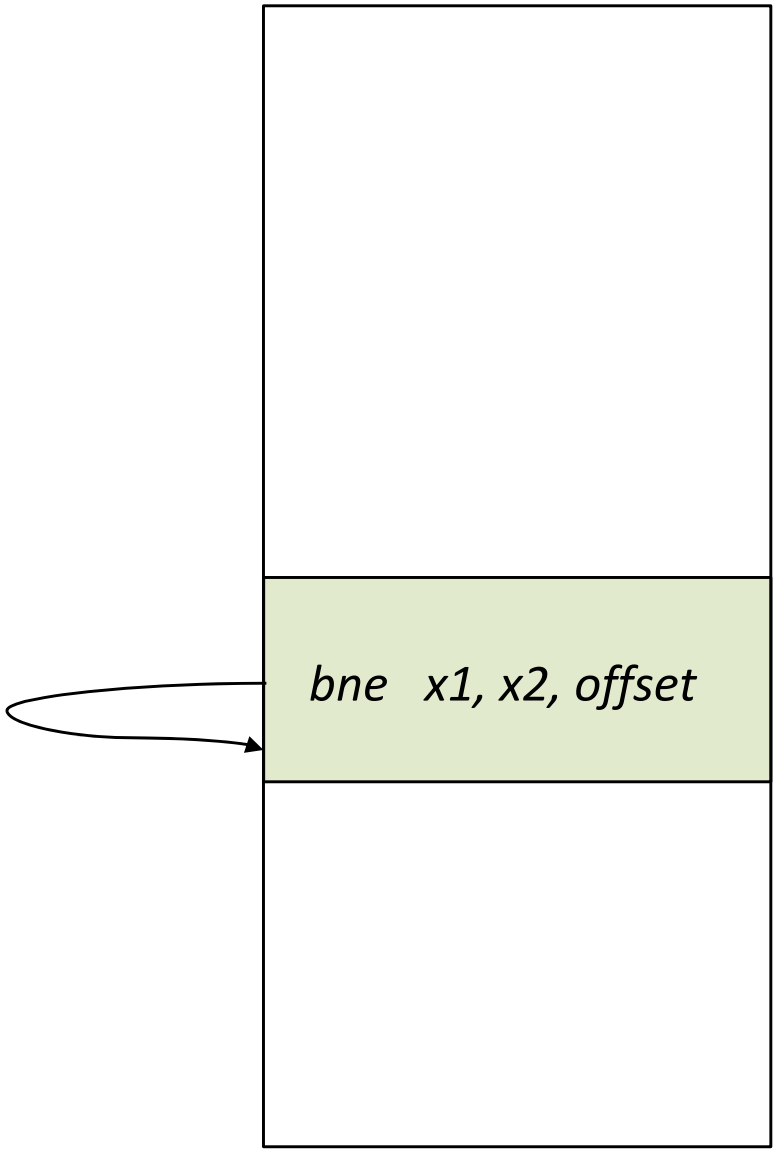


PC-Relative Branching

- Idea: use the PC as the base register
  - Target address = PC + offset
  - Don't have to specify a base register in the instruction encoding (because the PC is always the base register)
  - That gives you full 13 bits to hold the offset
  - Might want branch forward in instruction stream, or might want to branch back
    - Make the offset a signed value
      - -4096 to 4095

PC-Relative Branching

- Can branch back or ahead up to about 1,000 instructions from where you're executing now



The Simulator

- $ Sim --help
- usage: CSE 410 RISC-V Simulator [-h] [-b] [-d] [-D <disassembler path>]
- [-e ENTRYPOINT] [-o] [-p] [-t]
- codefiles [codefiles ...]

- optional arguments:
- -h, --help show this help message and exit
- -b, --byteaddressable
- Force byte addressable operation
- -d, --debugger Start execution in debugger
- -D <disassembler path>
- Path to C code disassembler executable
- -e ENTRYPOINT, --entrypoint ENTRYPOINT
- Start execution at entrypoint
- -o, --objectcode Input files contains object code, rather than
- assembler
- -p, --pcrelative Force pc relative branch operation
- -t, --trace Cause CPU to print instructions as executed