EEL 4713C Digital Computer Architecture – Spring 2026
Homework 1

Due Date: By the end of Feb 26, 2026
Total Points: 100

1. Background and Motivation

In this homework, you will study dynamic register pressure in a RISC-V processor by analyzing the execution behavior of a real workload. Register pressure is a fundamental concept in both computer architecture and compiler design. It describes how many architectural registers are simultaneously "in use" during execution, and it directly affects:

- Register file size and access cost
- Compiler register allocation and spilling
- Instruction scheduling and overall performance

In this assignment, register pressure is defined using dynamic liveness, measured from an execution trace of a RISC-V program. You will build a lightweight analysis tool that decodes binary instructions from an execution trace and computes register pressure using a backward liveness analysis.

2. Architectural Assumptions

Unless otherwise specified, assume the following:

- ISA: RISC-V RV32I
- Number of architectural registers: 32 (x0–x31)
- Register x0 is hardwired to zero and must NOT be counted toward register pressure
- In-order execution
- Each instruction completes in one cycle (cycle index = instruction index)
- You do NOT need to simulate register or memory values

3. Provided Execution Trace (Binary Instructions)

You will be given a pre-generated execution trace as a plain-text file.

- Each line corresponds to exactly one cycle
- Each line records one executed instruction

Trace format:

<cycle> <pc> <inst_bin>

Where:

- cycle: integer starting from 0
- pc: program counter in hexadecimal
- inst_bin: 32-bit binary string (exactly 32 characters of 0/1)

Example (illustrative only):

0 0x00000100 00000000011100110000001010110011
1 0x00000104 00000000010000101010010000000011
2 0x00000108 00000000000100101000001010010011
3 0x0000010c 00000000100000101010011000100011
4 0x00000110 11111111000000010100011101110011

4. Assembly Shown Only for Demonstration

For explanation purposes, examples in this handout may show instructions in assembly form.
However, the actual graded input trace contains ONLY binary instructions (inst_bin).

Example demonstration (NOT the real trace format):

cycle pc assembly
0 0x100 add x5, x6, x7
1 0x104 lw x8, 4(x5)
2 0x108 addi x5, x5, 1
3 0x10c sw x8, 8(x5)
4 0x110 beq x5, x0, -4

Your tool must decode the binary instructions by itself.

5. Part A – Instruction Decoder / Disassembler (Required)

You must implement an instruction decoder (using either python or C) that takes a 32-bit binary instruction and extracts the information needed for register pressure analysis.

Strictly speaking, this is a disassembler (binary → assembly-level fields).

For each instruction i, your tool must identify:

- Instruction type

- Destination register (if any)
- Source registers (if any)

You do NOT need to print full assembly, but you must correctly identify register reads and writes.

5.1 Supported Instruction Subset (RV32I)

Your decoder must support at least the following instructions:

R-type:

- *add, sub, and, or, xor, sll, srl, sra, slt, sltu*

I-type:

- *addi, andi, ori, xori, slti, sltiu*
- *lw*
- *jalr*

S-type:

- *sw*

B-type:

- *beq, bne, blt, bge, bltu, bgeu*

U-type:

- *lui, auipc*

J-type:

- *jal*

You may assume the trace only contains instructions from this subset.

5.2 Register Use / Definition Rules

Use the following rules to determine uses(i) and defs(i):

- R-type:
  defs = {rd}, uses = {rs1, rs2}
- I-type ALU:
  defs = {rd}, uses = {rs1}

- lw:
  defs = {rd}, uses = {rs1}
- sw:
  defs = {}, uses = {rs1, rs2}
- Branch (B-type):
  defs = {}, uses = {rs1, rs2}
- lui:
  defs = {rd}, uses = {}
- auipc:
  defs = {rd}, uses = {}
- jal:
  defs = {rd}, uses = {}
- jalr:
  defs = {rd}, uses = {rs1}

Note:

- x0 may appear in uses/defs but must not be counted in pressure.

6. Part B – Register Pressure via Dynamic Liveness (Required)

Register pressure is defined as the number of live architectural registers at a given cycle. A register is live at cycle i if the value currently stored in the register will be read by some future instruction before being overwritten.

6.1 Backward Liveness Algorithm

Let LIVE be the set of registers live after the current instruction.

Process the trace backward from the last cycle to cycle 0:

$LIVE(i) = (LIVE(i+1) - defs(i)) \cup uses(i)$

Register pressure at cycle i:

$pressure(i) = |LIVE(i)|$, excluding x0

Backward traversal is required for correctness.

7. Required Outputs

Your tool must report:

1. Average register pressure over all cycles
2. Maximum register pressure and the cycle where it occurs
3. Minimum register pressure

Bonus:

- Output a file containing per-cycle pressure:
  cycle, pressure

8. Analysis Questions (Short Answer)

Answer the following in your report:

1. Which regions of execution show the highest register pressure, and why?
2. Why must register x0 be excluded from register pressure analysis?
3. If the processor had only 16 registers, how many cycles would exceed this limit? What does this imply?

9. Submission Requirements

Submit:

1. Source code (.py or .c) of the analysis tool
    - The first command line argument is the path of the trace file
2. Short report (in PDF) including:
    - Summary statistics based on your tool implemented
    - Answers to analysis questions

10. Grading

- Instruction decoder correctness: 40%
- Liveness and pressure computation: 40%
- Report clarity and correctness: 20%
- The bonus question in (7) is worth an additional 5%. The total score for HW1 is capped at 100%
- For grading purposes, evaluation will be conducted using a different trace following the same format

Appendix A – RV32I Instruction Encoding Reference

This appendix summarizes the bit-level encoding of the RV32I instruction subset required for Homework 1. You may use this reference directly when implementing your instruction decoder. All instructions are 32 bits wide. Bit positions are numbered from bit 31 (most significant) to bit 0 (least significant).

R-type Instructions (e.g., add, sub, and, or, xor, sll, srl, sra)

Bit fields:

bits 31–25 : funct7
bits 24–20 : rs2
bits 19–15 : rs1
bits 14–12 : funct3
bits 11–7 : rd
bits 6–0 : opcode

Opcode:
0110011

Examples:

- add : funct3 = 000, funct7 = 0000000
- sub : funct3 = 000, funct7 = 0100000
- and : funct3 = 111, funct7 = 0000000
- or : funct3 = 110, funct7 = 0000000

Register usage:

- defs = {rd}
- uses = {rs1, rs2}

I-type Instructions (ALU-immediate, loads, jalr)

Bit fields:

bits 31–20 : immediate[11:0]
bits 19–15 : rs1
bits 14–12 : funct3
bits 11–7 : rd

bits 6–0 : opcode

Opcodes:

- ALU-immediate (addi, andi, ori, xori, slti, sltiu): 0010011
- Load word (lw): 0000011
- jalr: 1100111

Immediate:

- 12-bit signed immediate
- Must be sign-extended to 32 bits

Register usage:

- defs = {rd}
- uses = {rs1}

S-type Instructions (store word: sw)

Bit fields:

bits 31–25 : immediate[11:5]
bits 24–20 : rs2
bits 19–15 : rs1
bits 14–12 : funct3
bits 11–7 : immediate[4:0]
bits 6–0 : opcode

Opcode:
0100011

Immediate:

- 12-bit signed immediate constructed from bits [31–25] and [11–7]
- Must be sign-extended

Register usage:

- defs = {}
- uses = {rs1, rs2}

B-type Instructions (branches: beq, bne, blt, bge, bltu, bgeu)

Bit fields:

bit 31 : immediate[12]
bits 30–25 : immediate[10:5]
bits 24–20 : rs2
bits 19–15 : rs1
bits 14–12 : funct3
bits 11–8 : immediate[4:1]
bit 7 : immediate[11]
bits 6–0 : opcode

Opcode:
1100011

Immediate:

- 13-bit signed immediate (with implicit bit 0 = 0)
- Must be sign-extended
- You do NOT need to compute the branch target address for this homework

Register usage:

- defs = {}
- uses = {rs1, rs2}

U-type Instructions (lui, auipc)

Bit fields:

bits 31–12 : immediate[31:12]
bits 11–7 : rd
bits 6–0 : opcode

Opcodes:

- lui: 0110111
- auipc: 0010111

Immediate:

- Upper 20 bits, shifted left by 12
- Sign extension is not required for liveness analysis

Register usage:

- defs = {rd}
- uses = {}

J-type Instructions (jal)

Bit fields:

bit 31 : immediate[20]
bits 30–21 : immediate[10:1]
bit 20 : immediate[11]
bits 19–12 : immediate[19:12]
bits 11–7 : rd
bits 6–0 : opcode

Opcode:
1101111

Immediate:

- 21-bit signed immediate
- Must be sign-extended
- You do NOT need to compute the jump target address

Register usage:

- defs = {rd}
- uses = {}

Appendix B – Sample Trace and Expected Register Pressure Output

This appendix provides a small example trace and the expected register pressure results. This example is for illustration and self-checking only; it is NOT the actual graded trace.

Sample Input Trace (Binary)

cycle pc inst_bin
0 0x00000100 00000000011100110000001010110011
1 0x00000104 00000000010000101010010000000011
2 0x00000108 00000000000100101000001010010011
3 0x0000010c 00000000100000101010011000100011
4 0x00000110 11111110000000101000111011100011

Decoded Instructions (for explanation only)

cycle instruction
0 add x5, x6, x7
1 lw x8, 4(x5)
2 addi x5, x5, 1
3 sw x8, 8(x5)
4 beq x5, x0, -4

Register Uses and Definitions

cycle uses defs
0 x6, x7 x5
1 x5 x8
2 x5 x5
3 x8, x5 -
4 x5, x0 -

(Note: x0 is ignored in pressure counting.)

Backward Liveness Analysis

Starting from the last instruction:

Cycle 4:

LIVE = {x5}

Cycle 3:
LIVE = {x8, x5}

Cycle 2:
LIVE = {x5}

Cycle 1:
LIVE = {x5}

Cycle 0:
LIVE = {x6, x7}

Register Pressure per Cycle

cycle pressure
0 2
1 1
2 1
3 2
4 1

Summary Statistics

- Average register pressure = (2 + 1 + 1 + 2 + 1) / 5 = 1.4
- Maximum register pressure = 2
- Minimum register pressure = 1