# Operation of the Hardware: State Machine

CSE 410 22wi

Lecture 03.5

# Program vs. Instruction: Program

```
#define N 10
int main(int argc, char *argv[]) {
  int i, val[N];

  for (i=0; i<N; i++)  val[i] = 2*val[i-1]+7;

  sum = val[0];

  for (i=1; i<N; i++) sum += val[i];

  printf("sum = %d\n", sum);
  return 0;
}
```

- A program is a (complete) set of instructions
- The compiler can see them all
- The compiler can "reason" about the full program

- Can you spot a bug in this program?

# Program vs. Instruction: Program

```
#define N 10
int main(int argc, char *argv[]) {
  int i, val[N];

  for (i=0; i<N; i++)  val[i] = 2*val[i-1]+7;

  sum = val[0];

  for (i=1; i<N; i++) sum += val[i];

  printf("sum = %d\n", sum);
  return 0;
}
```

- In function ‘main’:
- error: ‘sum’ undeclared (first use in this function)
- sum = val[0];
- ^~~
- : each undeclared identifier is reported only once for each function it appears in

# Program vs. Instruction: Program

```
#define N 10
int main(int argc, char *argv[]) {
  int i, sum, val[N];

  for (i=0; i<N; i++)  val[i] = 2*val[i-1]+7;

  sum = val[0];

  for (i=1; i<N; i++) sum += val[i];

  printf("sum = %d\n", sum);
  return 0;
}
```

- There are limits to the compiler's ability to reason
- Can you spot a bug in this program?
  - There are (at least) two
- The (C) compiler doesn't

# Program vs. Instruction: Program

```
#define N 10
int main(int argc, char *argv[]) {
  int i, sum, val[N];

  for (i=0; i<N; i++)  val[i] = 2*val[i-1]+7;

  sum = val[0];

  for (i=1; i<N; i++) sum += val[i];

  printf("sum = %d\n", sum);
  return 0;
}
```

- Compilers for other languages **might** be able to find this bug

# Program vs. Instruction: Compiling

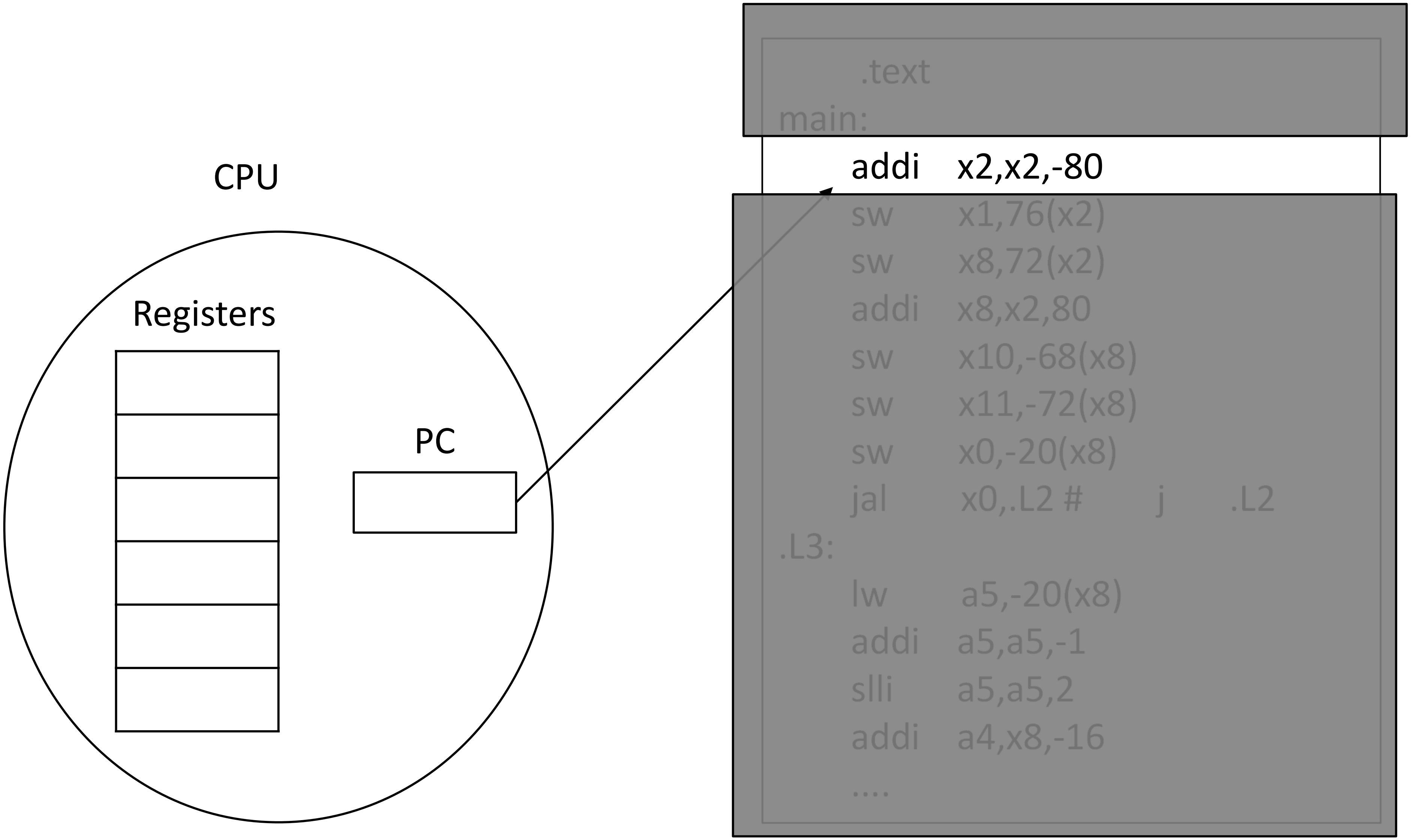
```
#define N 10
int main(int argc, char *argv[]) {
  int i, sum, val[N];

  for (i=0; i<N; i++)  val[i] = 2*val[i-1]+7;

  sum = val[0];

  for (i=1; i<N; i++) sum += val[i];

  printf("sum = %d\n", sum);
  return 0;
}
```

```
        .text
main:
        addi    x2,x2,-80
        sw      x1,76(x2)
        sw      x8,72(x2)
        addi    x8,x2,80
        sw      x10,-68(x8)
        sw      x11,-72(x8)
        sw      x0,-20(x8)
        jal     x0,.L2 #      j      .L2
.L3:
        lw      a5,-20(x8)
        addi    a5,a5,-1
        slli    a5,a5,2
        addi    a4,x8,-16
        ....
```

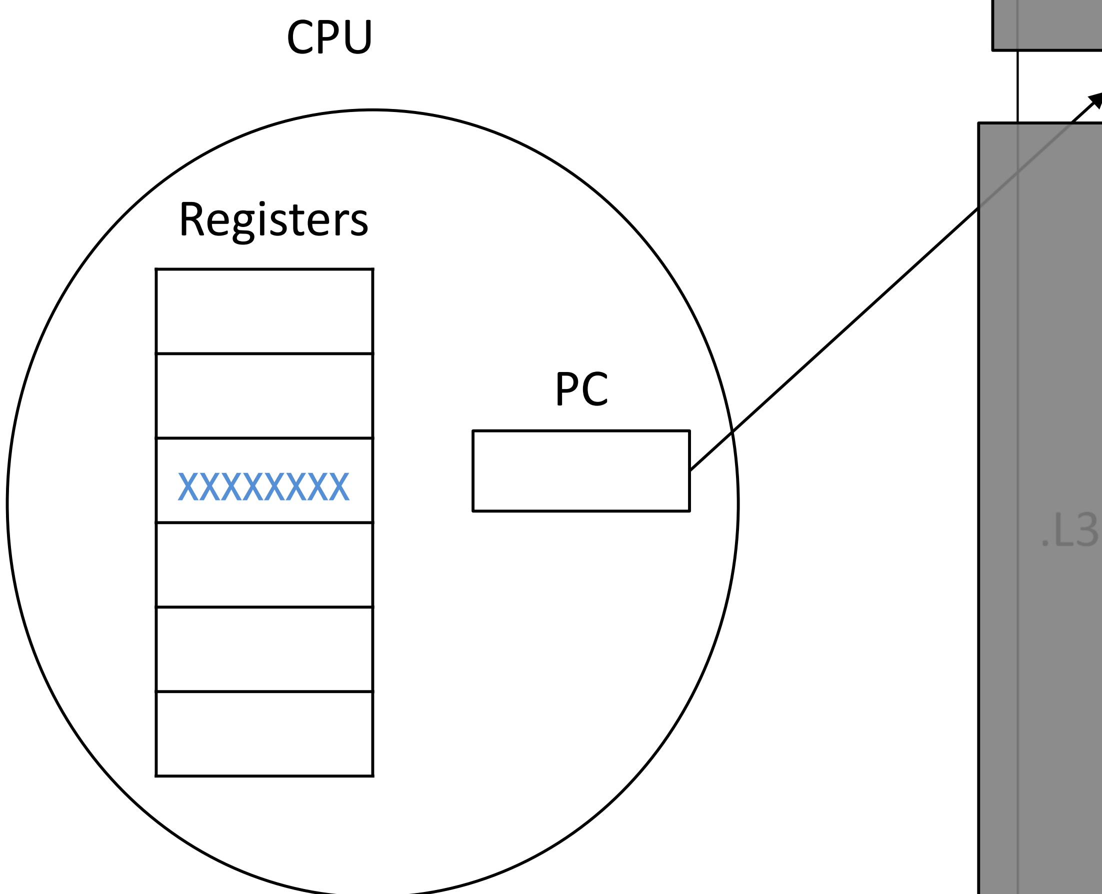
# Hardware: Instruction Execution

- The hardware is a "state machine"
  - It (behaves as though) it executes a single instruction at a time
  - The result of that execution depends only on the current "state" of the machine
    - The values of all registers, including the PC
    - The values in memory
  - The execution of an instruction doesn't depend on
    - instructions that were already executed (except for how they affected the current state)
    - what instructions will be executed in the future

# Program vs. Instruction: Execution

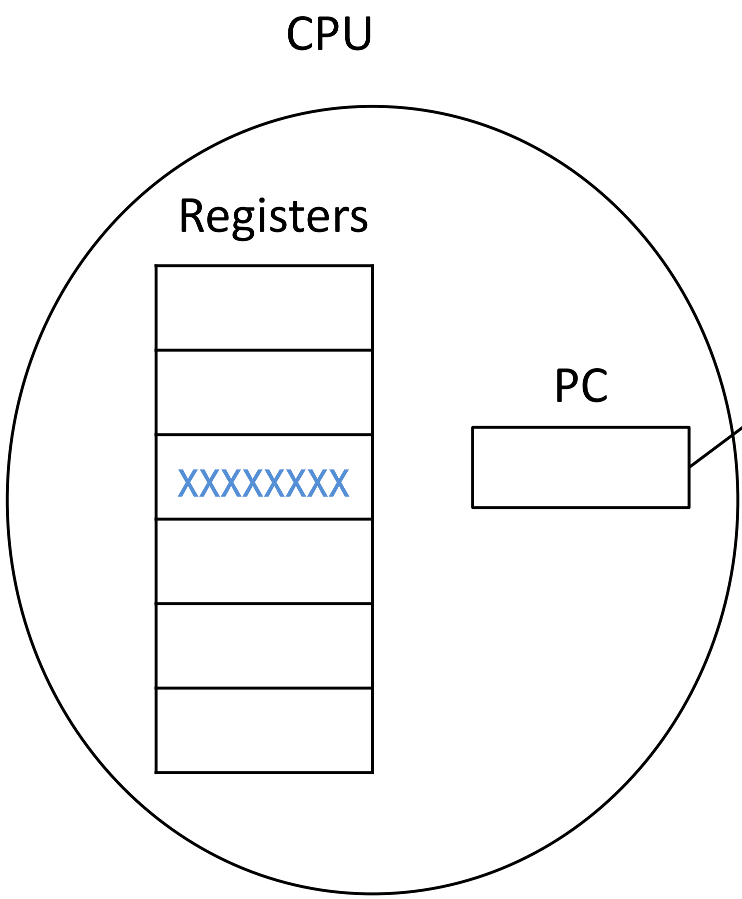


# Program vs. Instruction: Execution



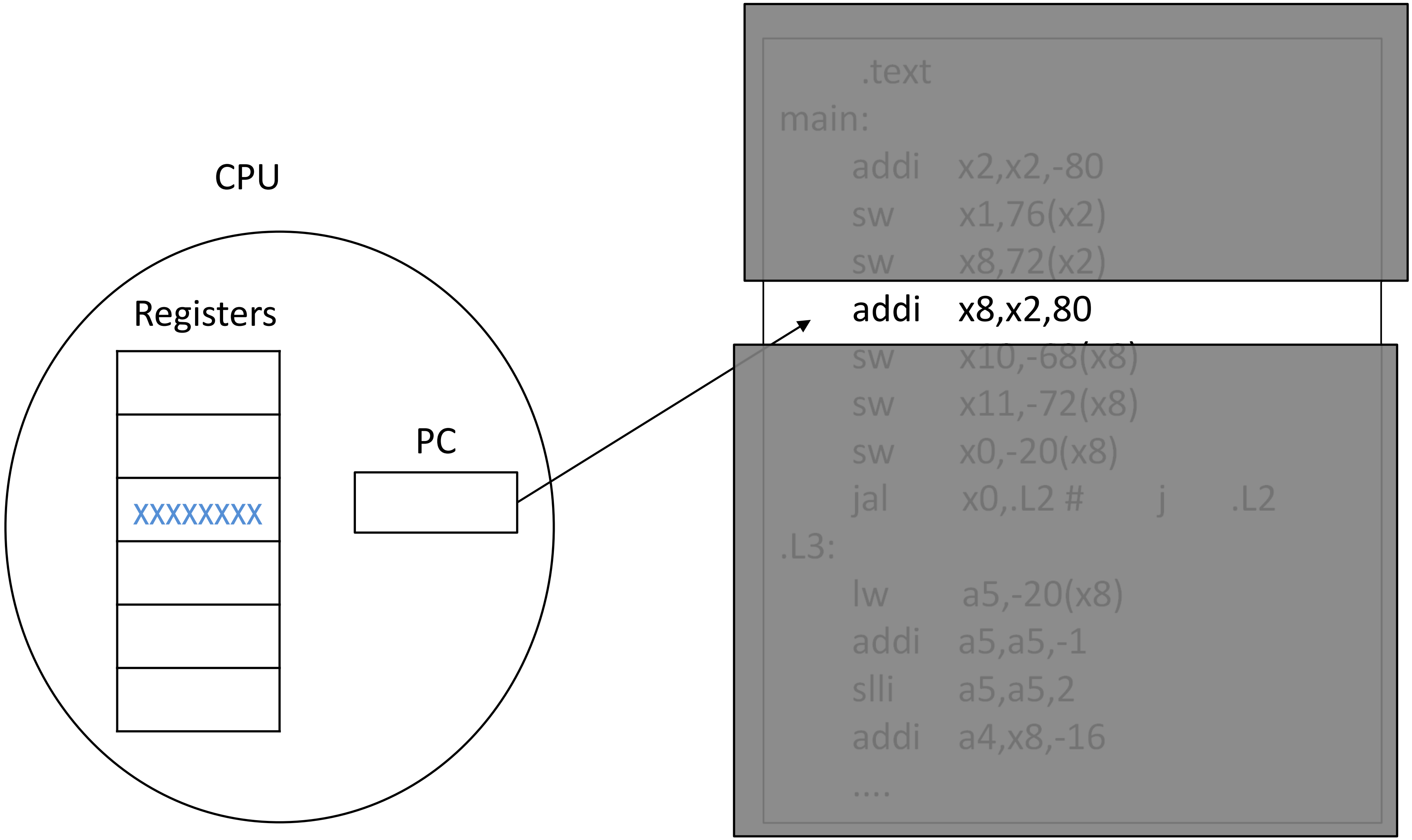

```
        .text
main:
        addi    x2,x2,-80
        sw      x1,76(x2)
        sw      x8,72(x2)
        addi    x8,x2,80
        sw      x10,-68(x8)
        sw      x11,-72(x8)
        sw      x0,-20(x8)
        jal     x0,.L2 #        j       .L2
.L3:
        lw      a5,-20(x8)
        addi    a5,a5,-1
        slli    a5,a5,2
        addi    a4,x8,-16
        ....
```

# Program vs. Instruction: Execution




```
        .text
main:
        addi    x2,x2,-80
        sw      x1,76(x2)
        sw      x8,72(x2)
        addi    x8,x2,80
        sw      x10,-68(x8)
        sw      x11,-72(x8)
        sw      x0,-20(x8)
        jal     x0,.L2 #      j      .L2
.L3:
        lw      a5,-20(x8)
        addi    a5,a5,-1
        slli    a5,a5,2
        addi    a4,x8,-16
        ....
```

# Program vs. Instruction: Execution



# Summary

- "Programs" are a static construct
  - Programmers, compilers, assemblers
- Program execution is a dynamic construct
- The hardware that performs the execution is a state machine
  - The idea of "program" is lost
  - All that is happening is execution of one instruction followed by execution of some next instruction

# Summary (cont.)

- Because the compiler can see the complete program, it might be able to detect errors that that won't be detected by the CPU
  - Example: array indexing error is just a lw instruction

- Because the hardware sees the dynamic state of the program, it might be able to detect errors that are hard or impossible to detect statically, by the compiler
  - Example: overflow

# Program vs. Instruction: Program

```
#define N 10
int main(int argc, char *argv[]) {
  int i, sum, val[N];

  for (i=0; i<N; i++)  val[i] = 2*val[i-1]+7;

  sum = val[0];

  for (i=1; i<N; i++) sum += val[i];

  printf("sum = %d\n", sum);
  return 0;
}
```

| N | Output |
|---|---|
| 10 | 16,298 |
| 20 | 16,777,060 |
| 30 | -226 |
| 40 | -296 |

*The RISC-V processor does* ***not*** *notice overflow. Some processors do, though. And the point is it could, because it sees the dynamic state of the program...*