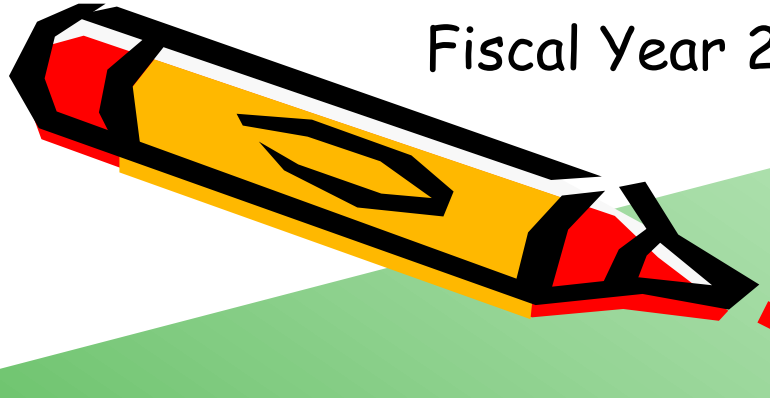
Fiscal Year 2024

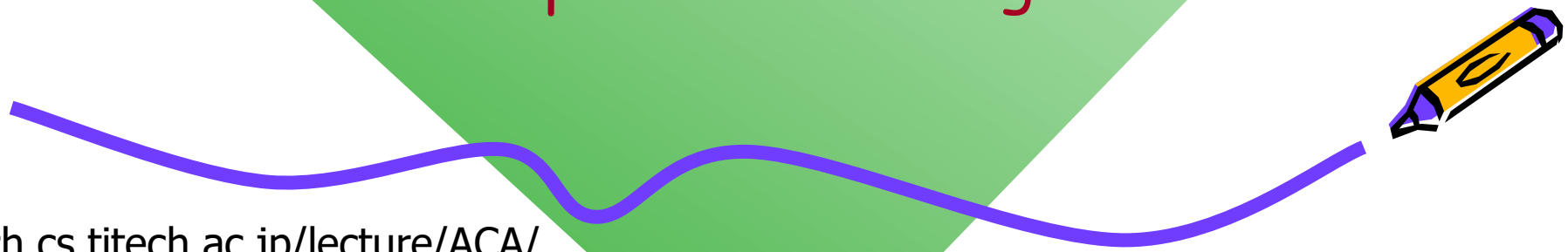
Ver. 2024-12-23b



Course number: CSC.T433
School of Computing,
Graduate major in Computer Science

Advanced Computer Architecture

5. Instruction Level Parallelism: Concepts and Challenges

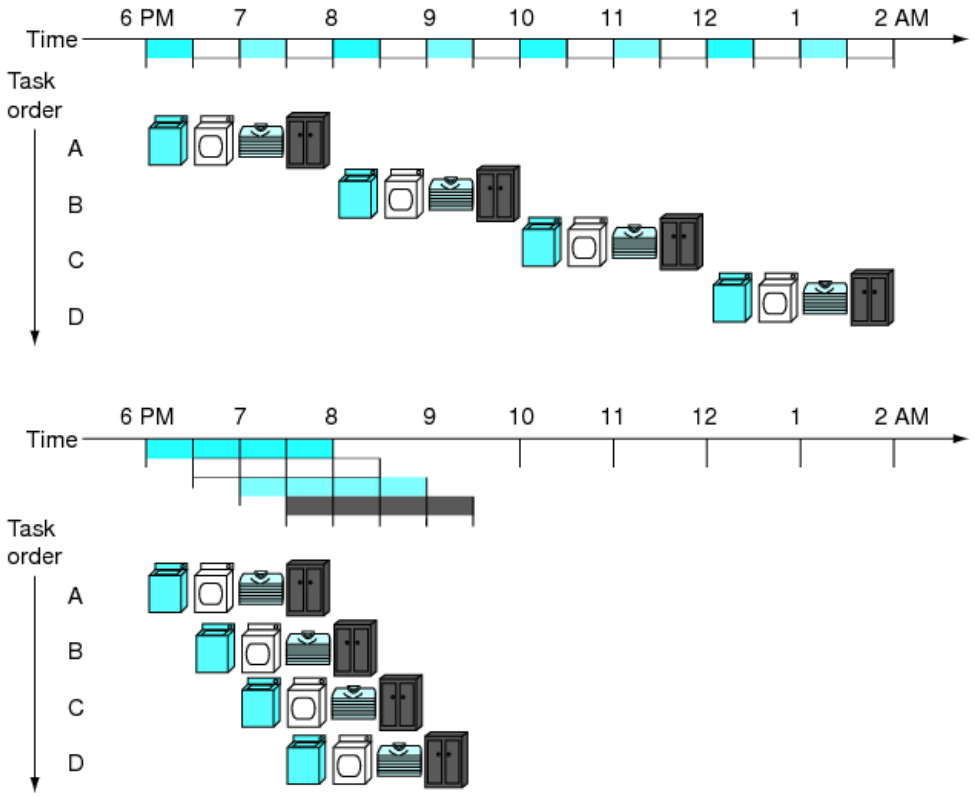


www.arch.cs.titech.ac.jp/lecture/ACA/
Room No. W8E-308, Lecture (Face-to-face)
Mon 13:30-15:10, Thr 13:30-15:10

Kenji Kise, Department of Computer Science
kise _at_ c.titech.ac.jp

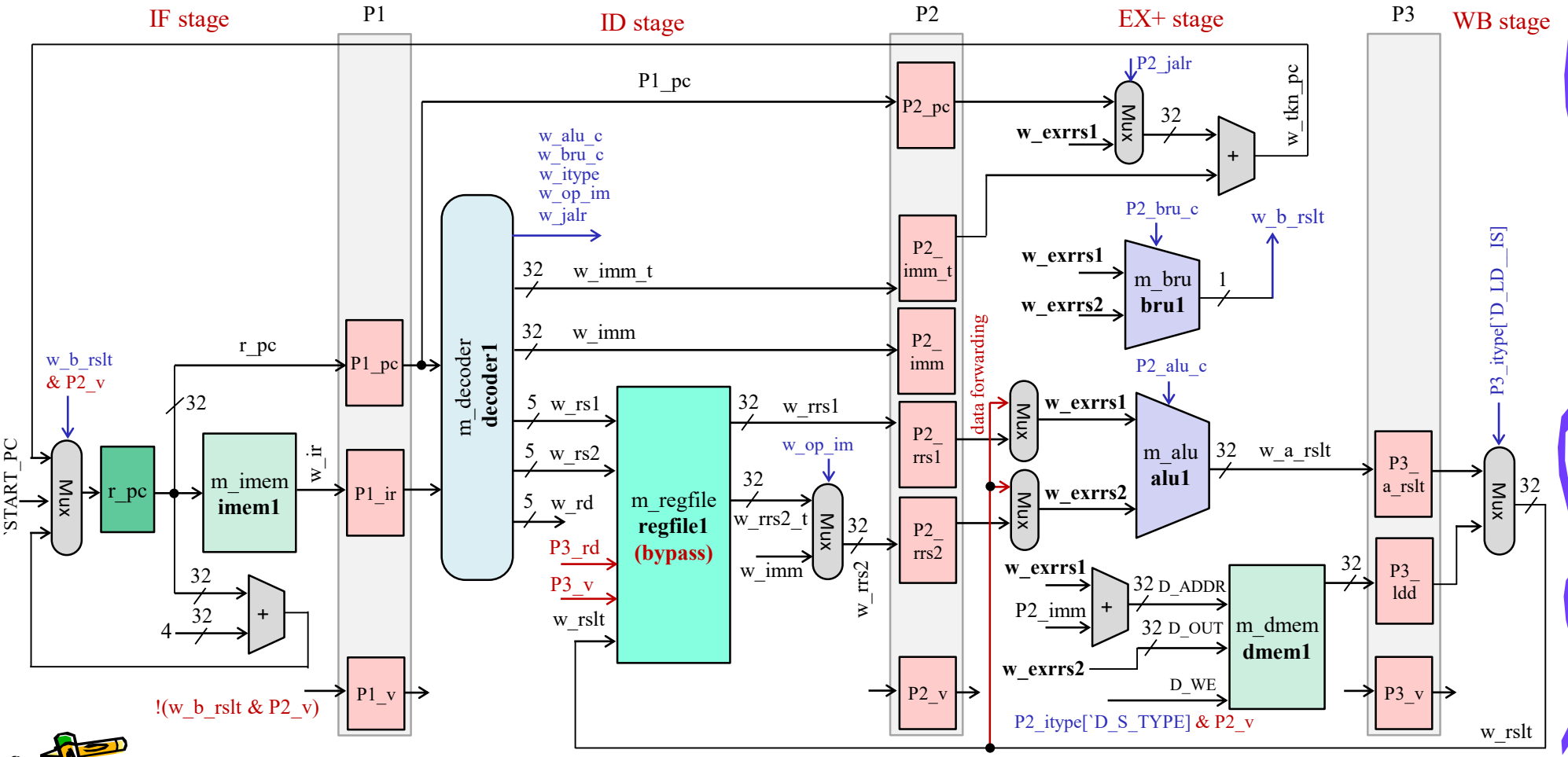
Single-cycle implementation and pipelining

- When the washing of load A is finished at 6:30 p.m., another washing of load B starts.
- Pipelined laundry takes 3.5 hours just using the same hardware resources. The cycle time is 30 minutes.
- What is the latency (execution time) of each load?



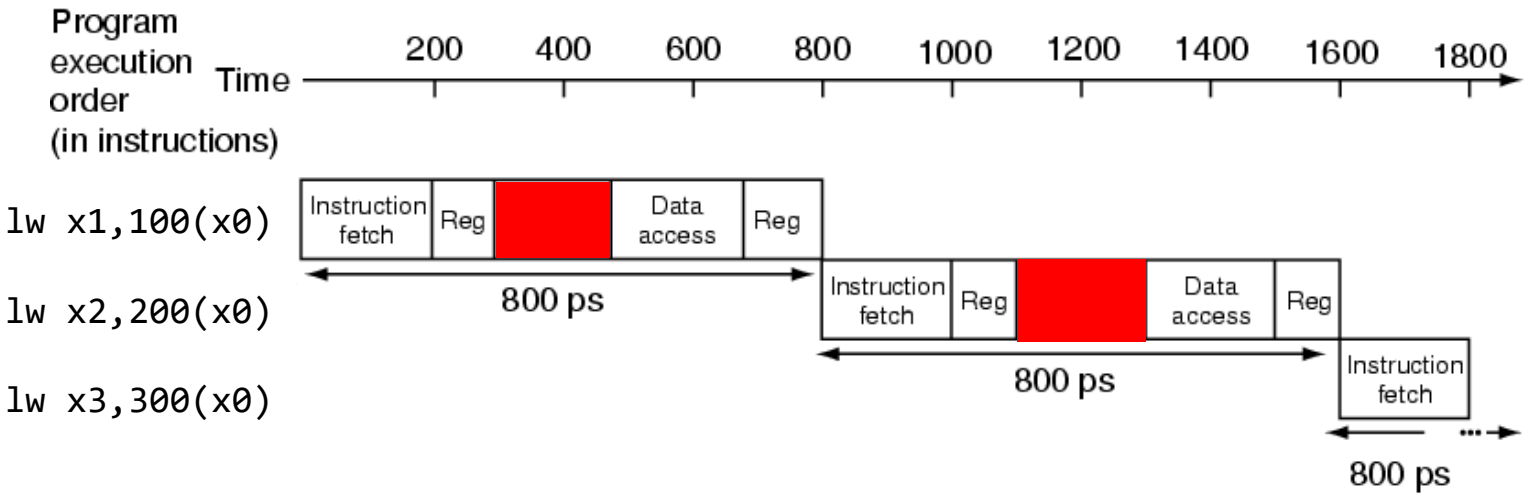
rvcore_4s : 4-stage pipelining processor with data forwarding

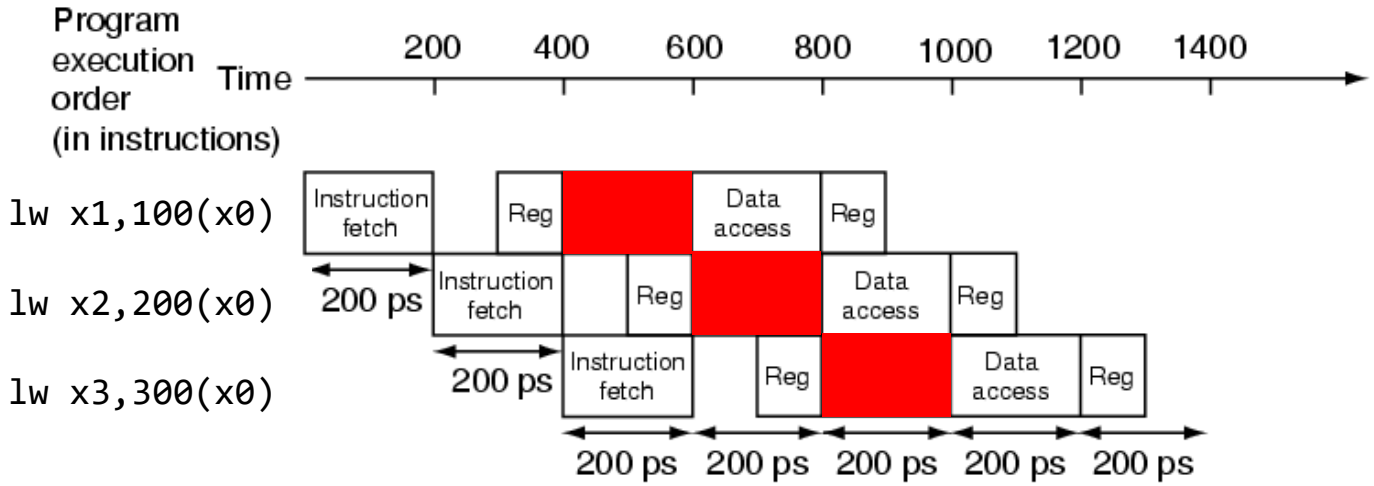
- The strategy is to separate the instruction fetch (IF) step, instruction decode (ID) step, and write back(WB) step, and other (EX, MA) steps. The first stage is named IF. The second stage is named ID. The third stage is named EX+. The last stage is named WB.



Single-cycle and pipelining processors

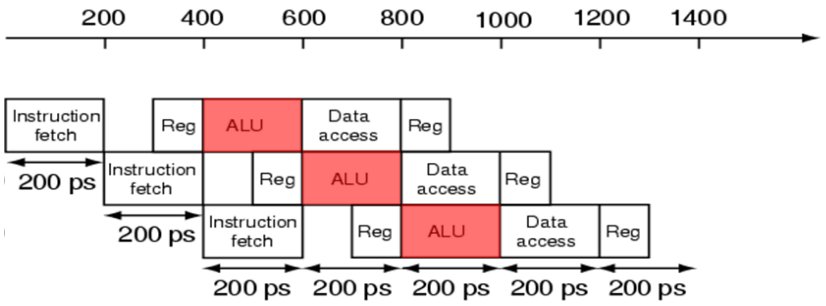
- The pipelining can improve ALU utilization to nearly 100%.



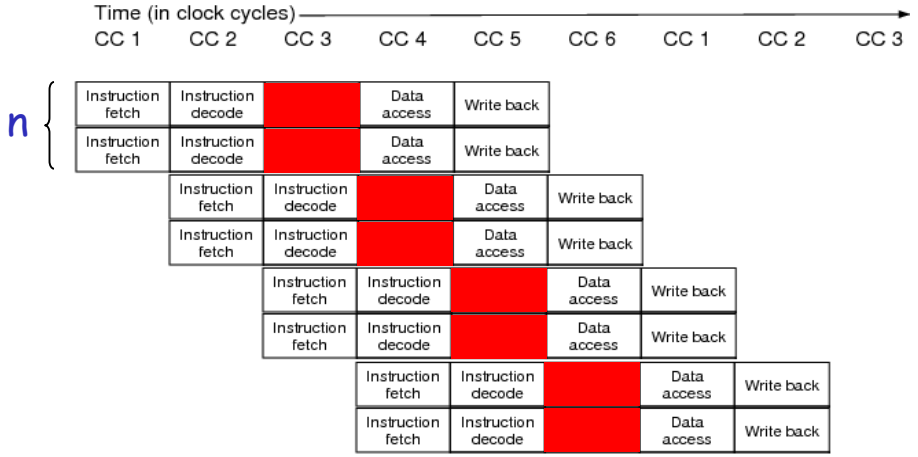


Scalar and Superscalar processors

- Scalar processor can execute at most one instruction per clock cycle by using one ALU.
  - IPC (Executed Instructions Per Cycle) is less than 1.
- Superscalar processor can execute more than one instruction per clock cycle by executing multiple instructions by using multiple pipelines.
  - IPC (Executed Instructions Per Cycle) can be more than 1.
  - using n pipelines is called n-way superscalar



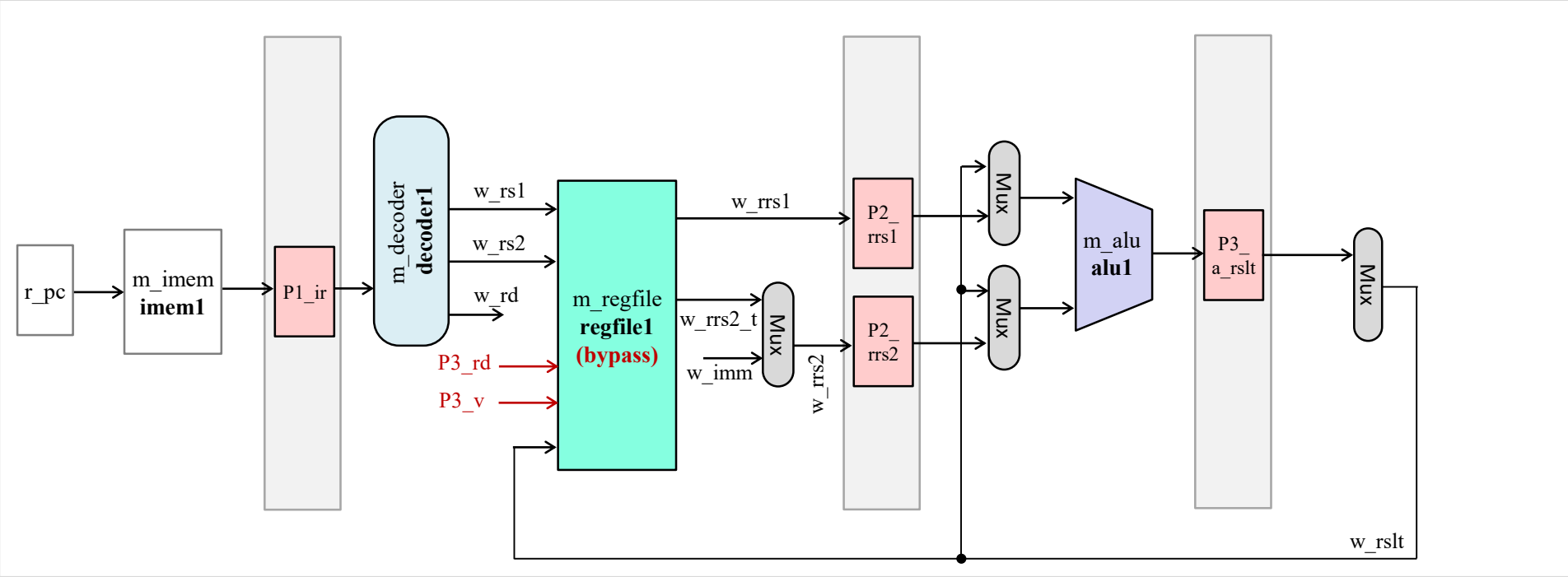
(a) pipeline diagram of scalar processor



(b) pipeline diagram of 2-way superscalar processor

Exercise 1

- Referring to the main datapath of rvcore_4s, design and draw a block diagram of a 2-way superscalar processor of 4-stage pipelining



Multi-Ported Memories (for FPGAs)

LVT (Live Value Tabele) design

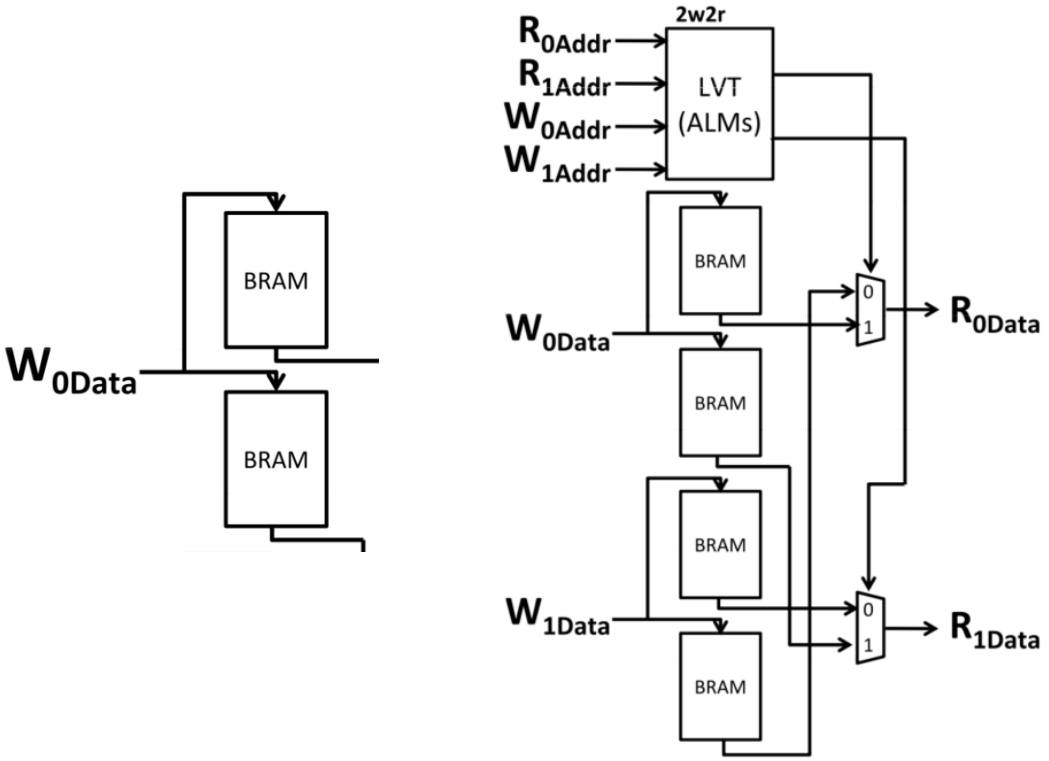


Figure 1: A 2W/2R Live Value Table (LVT) design.

[8] C. E. LaForest and J. G. Steffan. Efficient Multi-ported Memories for FPGAs. In *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '10, pages 41–50, New York, NY, USA, 2010. ACM.

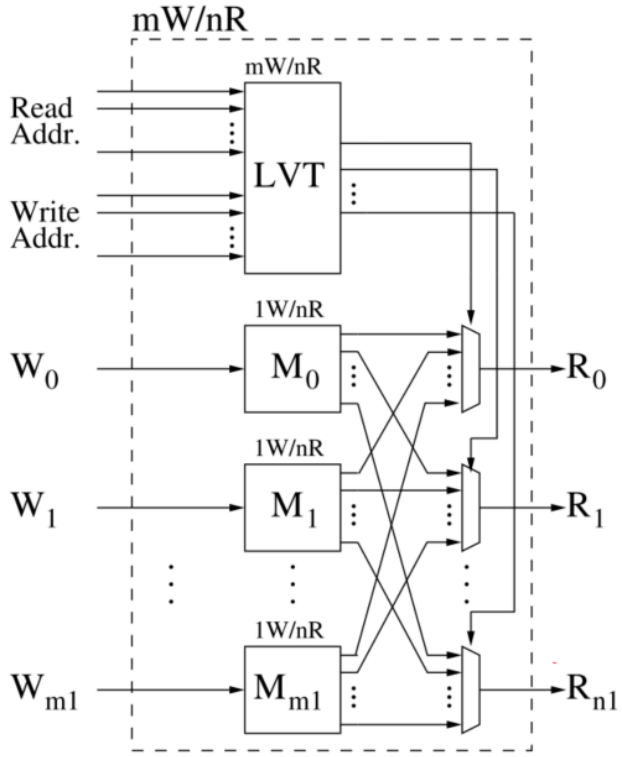


Figure 2: A generalized mW/nR memory implemented using a Live Value Table (LVT)

Exploiting Instruction Level parallelism (ILP)

- A superscalar has to handle some flows efficiently to exploit ILP
  - Control flow (control dependence)
    - To execute ***n*** instructions per clock cycle, the processor has to fetch at least ***n*** instructions per cycle.
    - The main obstacles are branch instruction like BNE, BEQ, ...
    - Prediction
    - Another obstacle is instruction cache
  - Register data flow (data dependence)
    - Out-of-order execution
      - Register renaming
      - Dynamic scheduling
  - Memory data flow
    - Out-of-order execution
    - Another obstacle is instruction cache

(5) RISC-V branch if not equal instructions (bne)

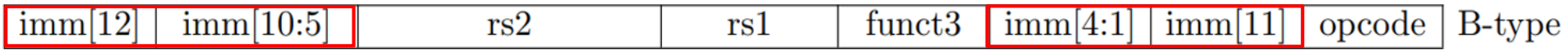
- RISC-V conditional branch instructions (bne, branch if not equal) :

```
 bne x4, x5, Lbl  # go to Lbl if x4!=x5
```

```
Ex:   if (i==j) h = i + j;
```

```
      bne x4, x5, Lbl1   # if (i!=j) goto Lbl1
      add x6, x4, x5     # h = i + j;
Lbl1: ...
```

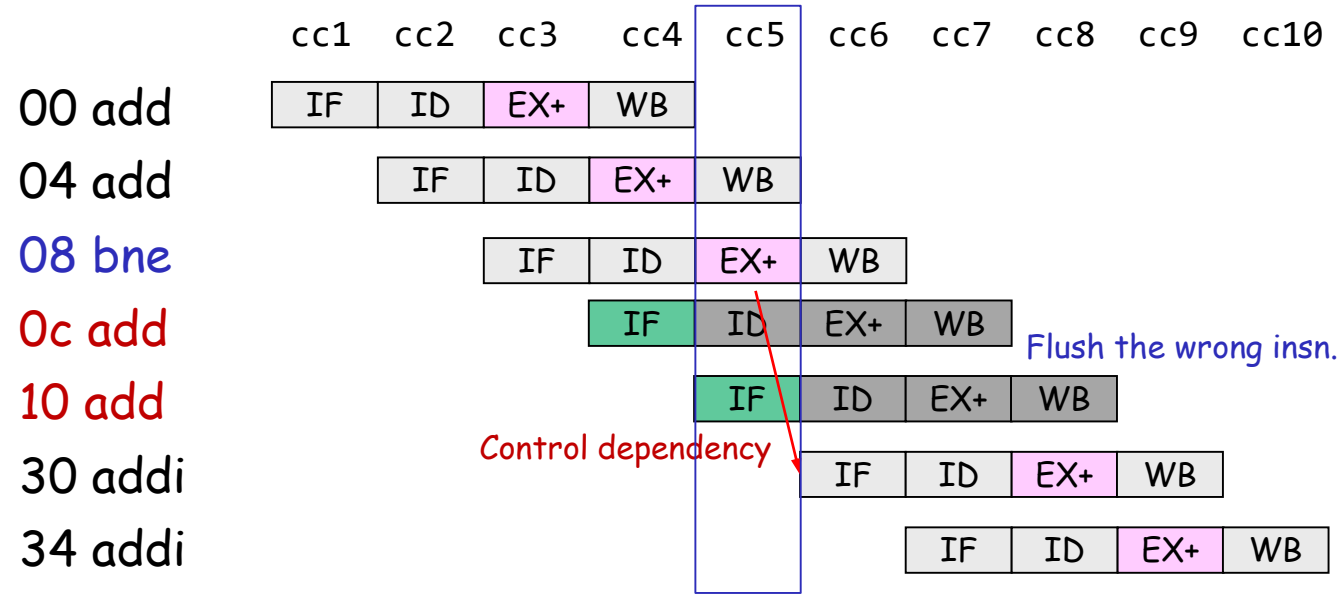
- Instruction Format (B-type):



- How is the branch destination address specified?

Why do branch instructions degrade IPC?

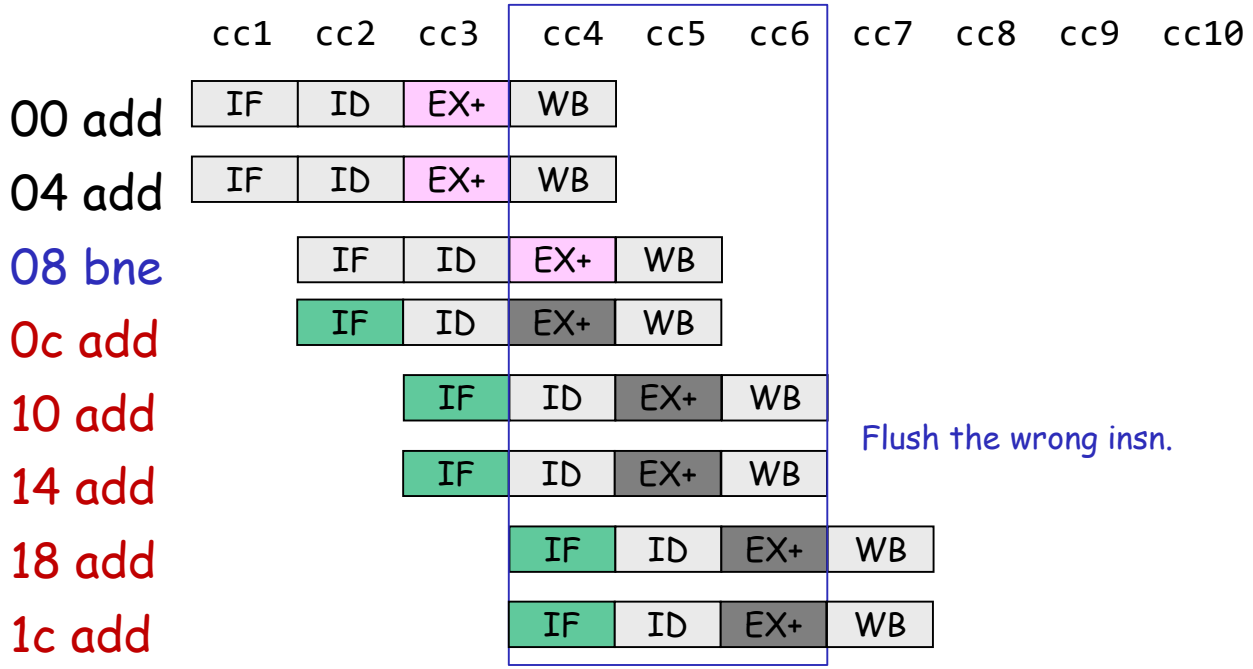
- Another approach is fetching the following instructions (0c add, 10 add) after a branch (bne) is fetched.
- When a branch (08 bne) is taken, the wrong instructions fetched (0c add, 10 add) are flushed.



four-stage pipelining processor executing instruction sequence with a taken branch

Why do branch instructions degrade IPC?

- Another approach is fetching the following instructions (an instruction at the next address and following ones) when a branch (bne) is fetched.
- When a branch (08 bne) is taken, the wrong instructions fetched are flushed.

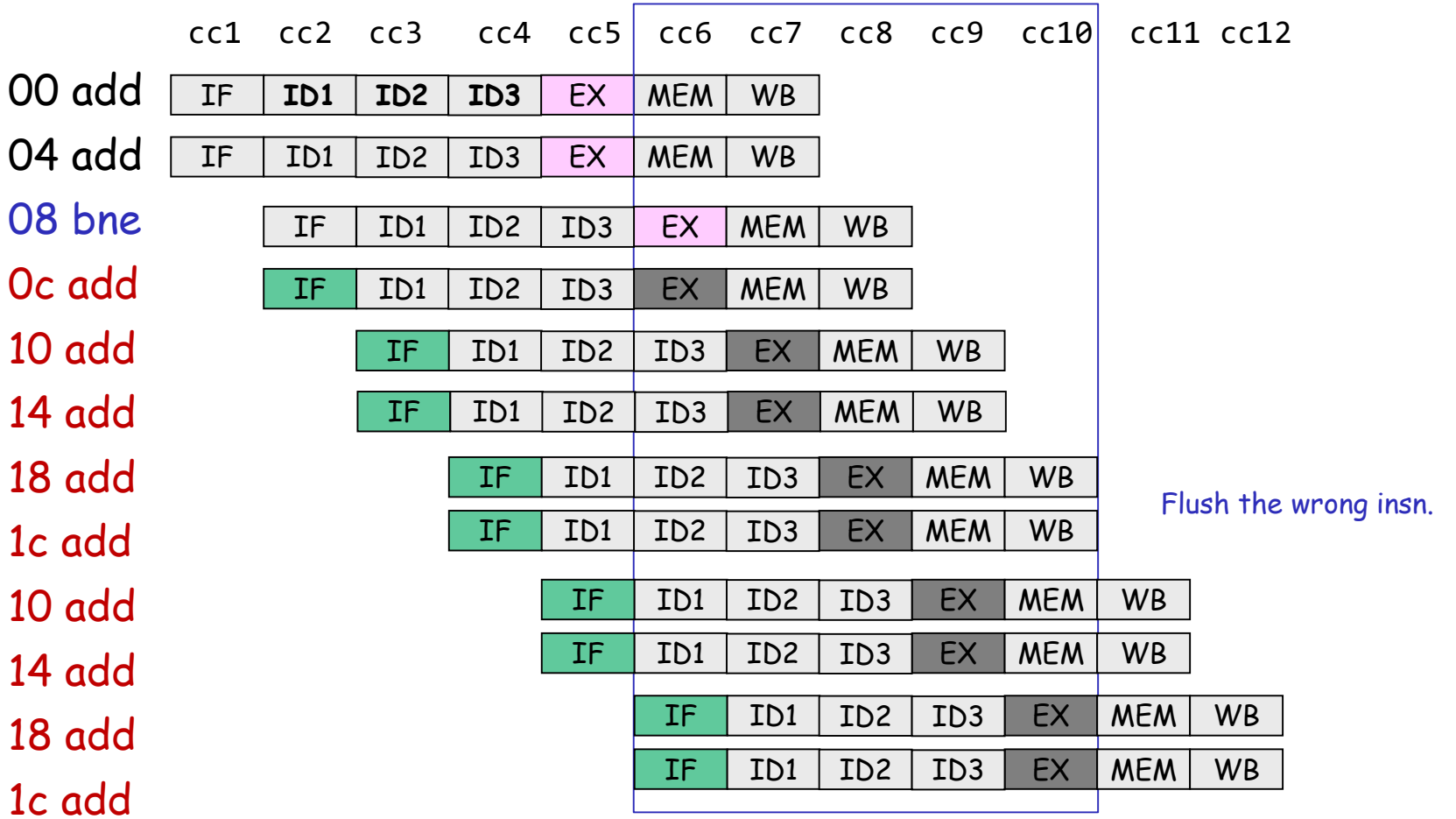


2-way superscalar processor executing instruction sequence with a branch

Because of the taken of a branch instruction, only one instruction is executed in cc4 and no instructions are executed in CC5 and CC6. This reduces the IPC.

Deeper pipeline with three ID stages

- Another approach is fetching the following instructions (an instruction at the next address and following ones) when a branch (bne) is fetched.



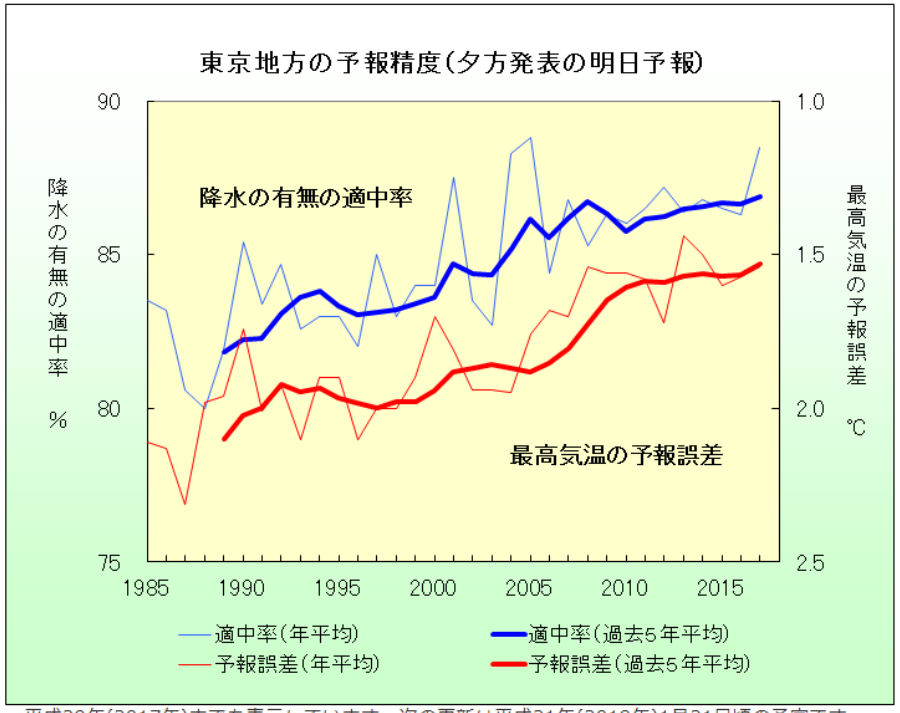
2-way superscalar adopting deeper pipeline executing instruction sequence with a branch

Hardware branch predictor

- A branch predictor is a digital circuit that tries to guess or predict which way (taken or untaken) a branch will go before this is known definitively.
  - A random predictor will achieve about a 50% hit rate because the prediction output is 1 (taken) or 0 (untaken).
  - Let's guess the accuracy.
    What is the accuracy of typical branch predictors for high-performance commercial processors?

Prediction Accuracy of weather forecasts

Predicting whether it will rain in the Tokyo area.
With the use of supercomputers, the prediction accuracy has gradually improved.
Based on data from several years ago, the accuracy of predicting the next day is around 85%.



平成29年(2017年)までを表示しています。次の更新は平成31年(2019年)1月31日頃の予定です。

年平均	北海道	東北	関東甲信	東海	北陸	近畿	中国	四国	九州北部	九州南部	沖縄	全国平均
明日	79	81	85	85	84	84	84	84	85	85	79	83
明後日	75	77	81	82	80	80	81	80	81	81	75	79
3日目	71	72	76	77	75	76	76	77	76	76	71	75
4日目	68	70	74	74	72	73	73	74	73	73	69	72
5日目	66	67	72	72	69	71	71	72	71	70	68	70
6日目	65	65	70	70	66	70	69	71	70	68	67	68
7日目	63	64	69	68	64	67	67	69	68	67	65	67
3～7日目平均	67	68	72	72	69	71	71	73	72	71	68	70

MLIT
Ministry of Land, Infrastructure, Transport and Tourism

国土交通省
気象庁
Japan Meteorological Agency

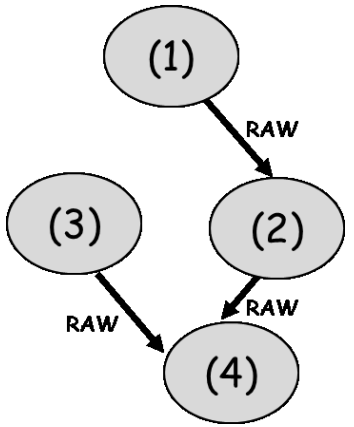
Tomorrow will be rainy?

Exploiting Instruction Level parallelism (ILP)

- A superscalar has to handle some flows efficiently to exploit ILP
  - Control flow (control dependence)
    - To execute ***n*** instructions per clock cycle, the processor has to fetch at least ***n*** instructions per cycle.
    - The main obstacles are branch instruction (BNE)
    - Prediction
    - Another obstacle is instruction cache
  - Register data flow (data dependence)
    - Out-of-order execution
      - Register renaming
      - Dynamic scheduling
  - Memory data flow
    - Out-of-order execution
    - Another obstacle is instruction cache

```
(1) add x5,x1,x2
(2) add x9,x5,x3
(3) lw  x4, 4(x7)
(4) add x8,x9,x4
```

```
(3) lw  x4, 4(x7)
(1) add x5,x1,x2
(2) add x9,x5,x3
(4) add x8,x9,x4
```



True data dependence

- Insn i writes a register that insn j reads, RAW (read after write)
- Program order must be preserved to ensure insn j receives the value of insn i.

```
   x3 = 10                    x3 = 10
   x5 = 2                     x3 = 10
   x3 = x3 x x5      (1)      x3 = 20
   x4 = x3 + 1       (2)      x3 = 20
i  x3 = x5 + 3       (3)      x3 = 5
j  x7 = x3 + R4      (4)      x3 = 5
```

wrong sequence reordering (3) and (4)

```
   x3 = 10                    x3 = 10
   x5 = 2                     x3 = 10
   x3 = x3 x x5      (1)      x3 = 20
   x4 = x3 + 1       (2)      x3 = 20
j  x7 = x3 + x4      (4)      x3 = 20
i  x3 = x5 + 3       (3)      x3 = 5
```

```
      20 = 10 x 2       (1)
      21 = 20 + 1       (2)
i     5  = 2  + 3       (3)
j     26 = 5  + 21      (4)
```

```
      20 = 10 x 2       (1)
      21 = 20 + 1       (2)
j     41 = 20 + 21      (4)
i     5  = 2  + 3       (3)
```

Output dependence

- Insn i and j write the same register, WAW (write after write)
- Program order must be preserved to ensure that the value finally written corresponds to instruction j.

```
  x3 = 10                    x3 = 10
  x5 = 2                     x3 = 10
i x3 = x3 x x5      (1)      x3 = 20
  x4 = x3 + 1       (2)      x3 = 20
j x3 = x5 + 3       (3)      x3 = 5
  x7 = x3 + R4      (4)      x3 = 5
```

wrong sequence reordering (1) and (3)

```
  x3 = 10                    x3 = 10
  x5 = 2                     x3 = 10
j x3 = x5 + 3       (3)      x3 = 5
  x4 = x3 + 1       (2)      x3 = 5
i x3 = x3 x x5      (1)      x3 = 20
  x7 = x3 + x4      (4)      x3 = 20
```

```
i 20 = 10 x 2       (1)
  21 = 20 + 1       (2)
j 5  = 2  + 3       (3)
  26 = 5  + 21      (4)
```

```
j 5  = 2  + 3       (3)
  6  = 5  + 1       (2)
i 20 = 10 x 2       (1)
  41 = 20 + 21      (4)
```

Antidependence

- Insn i reads a register that insn j writes, WAR (write after read)
- Program order must be preserved to ensure that i reads the correct value.

wrong sequence reordering (2) and (3)

```
   x3 = 10                    x3 = 10
   x5 = 2                     x3 = 10
   x3 = x3 x x5      (1)      x3 = 20
i  x4 = x3 + 1       (2)      x3 = 20
j  x3 = x5 + 3       (3)      x3 = 5
   x7 = x3 + x4      (4)      x3 = 5
```

```
   x3 = 10                    x3 = 10
   x5 = 2                     x3 = 10
   x3 = x3 x x5      (1)      x3 = 20
j  x3 = x5 + 3       (3)      x3 = 5
i  x4 = x3 + 1       (2)      x3 = 5
   x7 = x3 + x4      (4)      x3 = 5
```

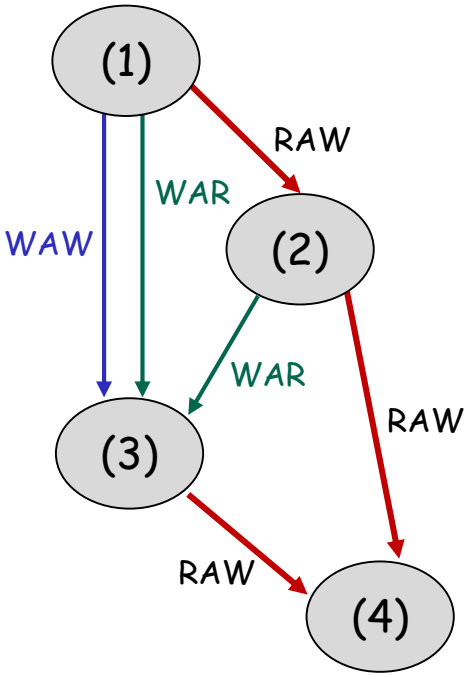
```
   20 = 10 x 2       (1)
i  21 = 20 + 1       (2)
j  5  = 2  + 3       (3)
   26 = 5  + 21      (4)
```

```
   20 = 10 x 2       (1)
j  5  = 2  + 3       (3)
i  6  = 5  + 1       (2)
   11 = 5  + 6       (4)
```

Data dependence and register renaming

- True data dependence (RAW)
- Name (false) dependences
  - Output dependence (WAW)
  - Antidependence (WAR)

```
x3 = x3 x x5    (1)
x4 = x3 + 1     (2)
x3 = x5 + 3     (3)
x7 = x3 + x4    (4)
```

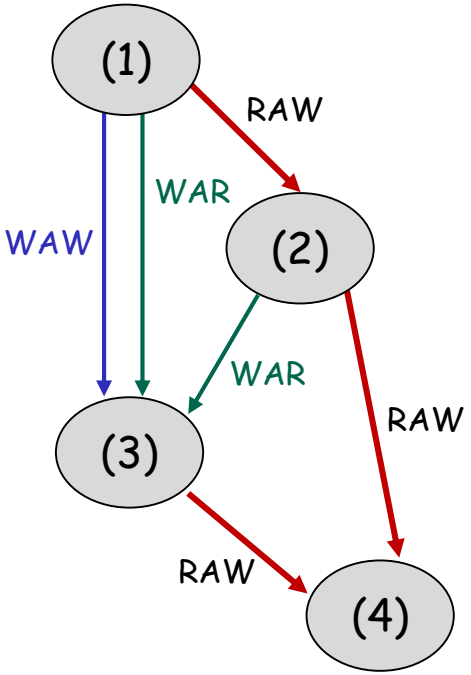


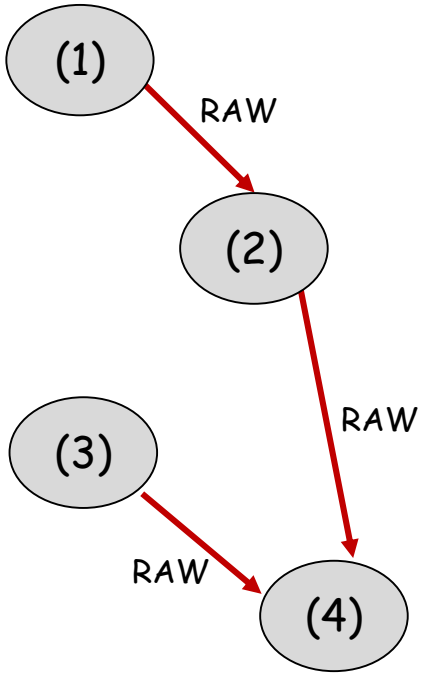
Data dependence and register renaming

- True data dependence (RAW)
- Name (false) dependences
  - Output dependence (WAW)
  - Antidependence (WAR)

```
x3 = x3 x x5    (1)
x4 = x3 + 1     (2)
x8 = x5 + 3     (3)
x7 = x8 + x4    (4)
```

```
x3 = x3 x x5    (1)
x4 = x3 + 1     (2)
x3 = x5 + 3     (3)
x7 = x3 + x4    (4)
```

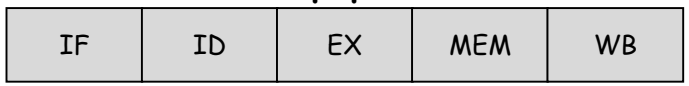




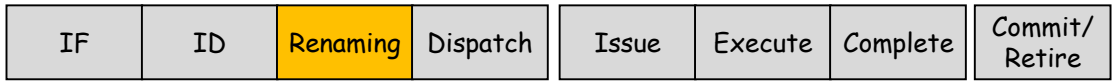
Hardware register renaming

- Logical registers (architectural registers) which are ones defined by ISA
  - x0, x1, … x31
- Physical registers
  - Assuming plenty of registers are available, p0, p1, p2, …
- A processor renames each register to eliminate false data dependency dynamically in the renaming stage

Typical instruction pipeline of scalar processor



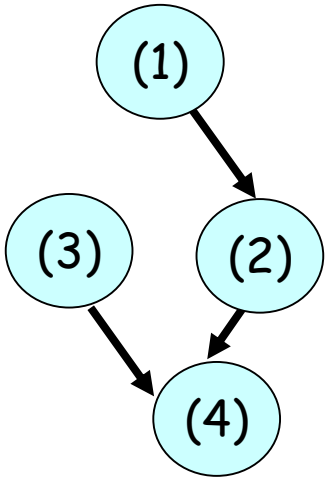
Typical instruction pipeline of high-performance superscalar processor



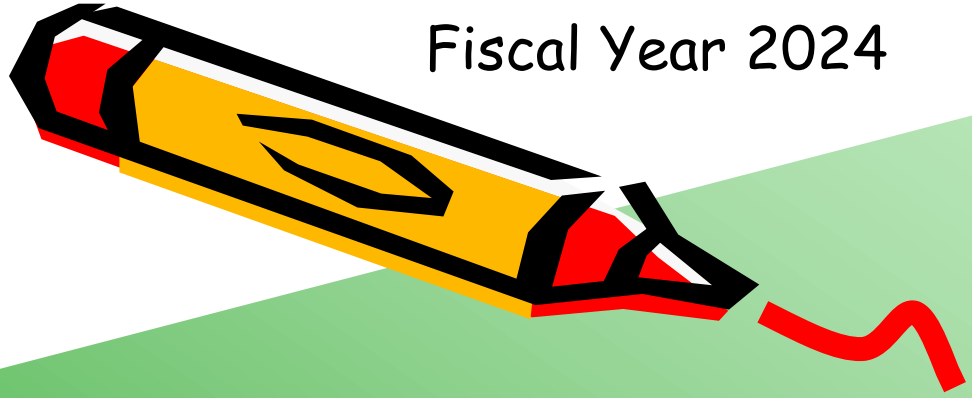
In-order and out-of-order (OoO) execution

- In in-order execution model, all instructions are executed in the order that they appear like (1), (2), (3), and (4).
  This can lead to unnecessary stalls.
  - Instruction (3) stalls waiting for insn (2) to go first, even though it does not have a data dependence.
- With out-of-order execution,
  - Using register renaming to eliminate output dependence and antidependence, just having true data dependence
  - Dynamic scheduling: insn (3) is allowed to be executed before the insn (2)
    - Tomasulo algorithm
      (IBM System/360 Model 91 in 1967)

```
x3 = x3 x x5    (1)
x4 = x3 + 1     (2)
x8 = x5 + 3     (3)
x7 = x8 + x4    (4)
```



Data flow graph



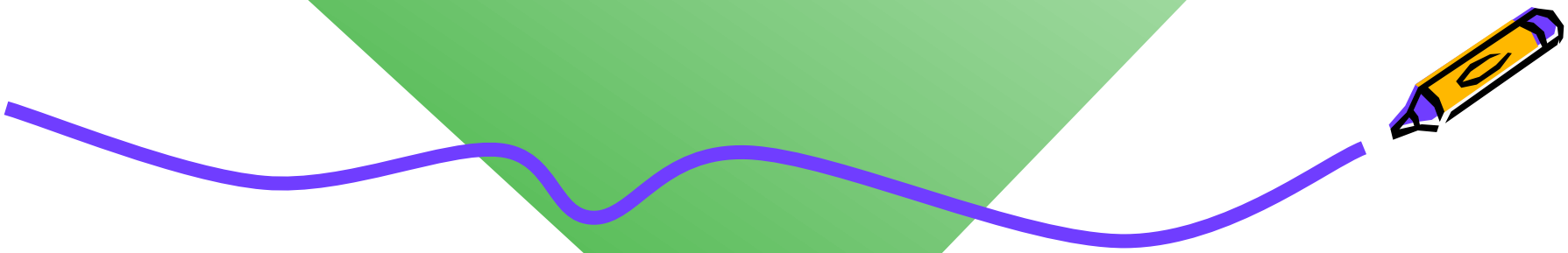
Fiscal Year 2024

Ver. 2024-12-23a

Course number: CSC.T433
School of Computing,
Graduate major in Computer Science

Advanced Computer Architecture

Mid-term report



Kenji Kise, Department of Computer Science
kise _at_ c.titech.ac.jp

Mid-term report

1. Please submit your mid-term report describing your answers to questions 1 and 2 in a PDF file via E-mail (kise [at] c.titech.ac.jp) by **January 9, 2024**
   - E-mail title should be "Report of Advanced Computer Architecture"
2. Please submit the report in 8 pages or less on A4 size PDF file, including the cover page.

3. You can discuss it with your colleague, but try to solve the questions yourself. Enjoy!

1. RISC-V assembly programming

- Write RISC-V assembly code asm1.s by hand compiling code1.c below. Use Venus RISC-V editor and simulator to show that the output of the code you wrote is correct.

```
int sum = 0;
int i, j;
for (i=1; i=<100; i=i+2)
  for (j=1; j=<100; j++) sum += (j+i);
```

code1.c

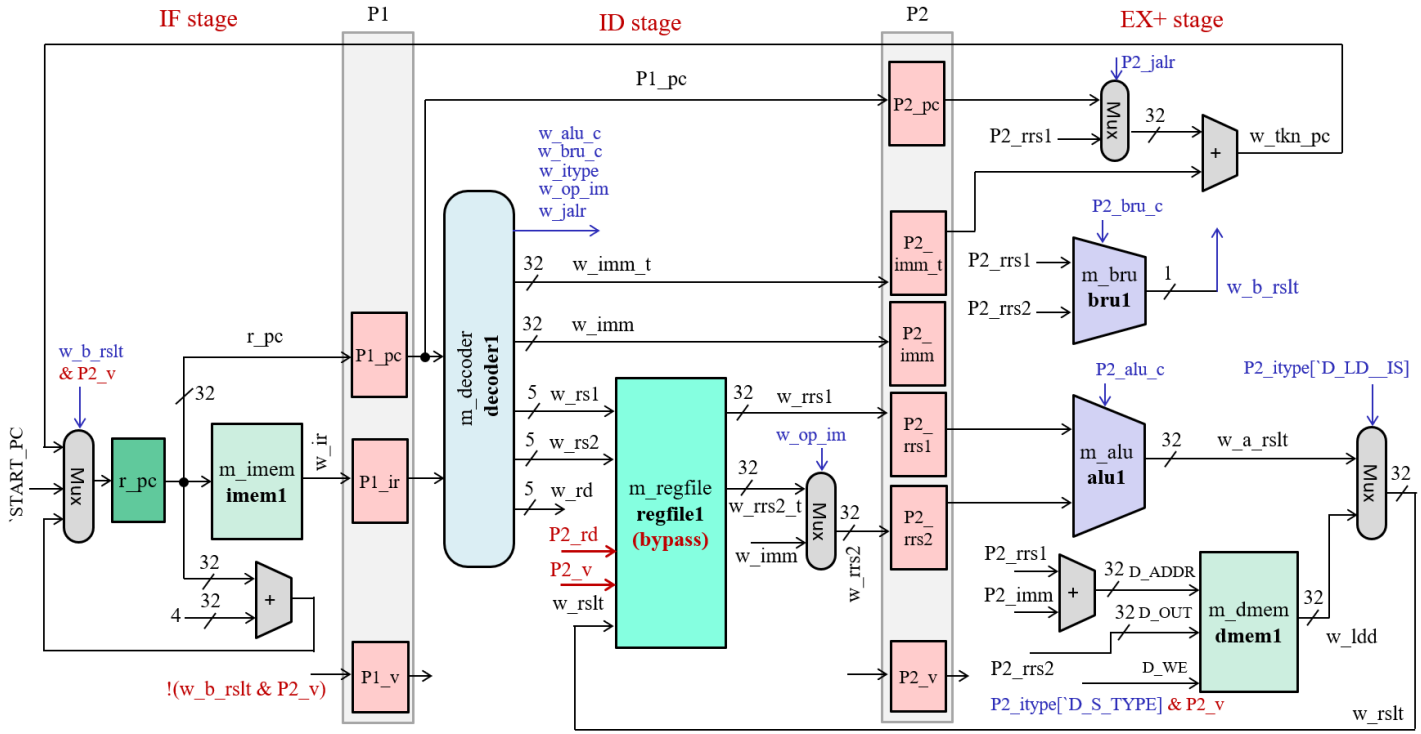
- Write RISC-V assembly code asm2.s by hand compiling code2.c below. Use Venus RISC-V editor and simulator to show that the output of the code you wrote is correct.

```
int A[200];
int sum = 0;
int i;
for (i=0; i<200; i++) A[i] = i + i;          /* initialize the array */
for (i=1; i<200; i++) A[i] = A[i-1] + A[i]; /* compute              */
for (i=0; i<200; i++) sum += A[i];           /* obtain the sum       */
```

code2.c

2. three-stage pipelining processor

- Describe a 3-stage pipelining scalar processor (rvcore_3s) in Verilog HDL. The report should include the description of module m_rvcore_3s.
- Verify the described code by compaing vefiry.txt generated by simulations of rvcore1 and rvcore_3s.



2. three-stage pipelining processor

- login the gateway server of ACRi room with your username
  - `ssh username@gw.acri.c.titech.ac.jp`
- login a compute server of ACRi room, select one among vs100, vs200, vs300, and vs400
  - `ssh vs300`
- copy the project directory to your working directory
  - `cd`
  - `mkdir -p aca`
  - `cd aca`
  - `cp -r /home/tu_kise/aca/rvcore_2s/ .`
  - `cp -r /home/tu_kise/aca/rvcore_3s/ .`
- simulate and test two-stage pipelining processor
  - `cd rvcore_2s`
  - `make`
  - `make run`
- implement your three-stage pipelining processor, simulate it, and vefity it
  - `cd ~/aca/rvcore_3s`
  - `emacs proc1.v` (please use your favorit text editor)
  - `make`
  - `make run`
  - `diff verify.txt ../rvcore_2s/verify.txt`