Green Computing Platforms

Step1. CA0602+0604: Pipeline, Superscalar and VLIW

http://archlab.naist.jp/Lectures/ARCH/ca0602_0604/ca060204e.pdf

Copyright © 2024 NAIST Y.Nakashima

Download the template and submit through UNIPA.

http://archlab.naist.jp/Lectures/ARCH/ca0602/ca0602e.docx

in http://archlab.naist.jp/Lectures

(1) Parallelization in CPU core (single program counter)(1960-1970)

	2016 IMAX	2012 EMAX	2008 LAPP	2006 OROCHI	1992 VPP	2002 Hyper Threading	2000 GPU	1997 DAISY	1964 Super Scalar	1965 Vector Processor	1976 VLIW	1982 Multithreading	1982 CGRA
1966 (TI ASC) Pipelining	○	○	○	○	○	○	○	○	○	○	○	○	○
1964 (CDC6600) Super Scalar				○		○			○				
1965 (ILLIAC IV) Vector Processor	○	○			○		○			○			
1976 (QA1) VLIW	○	○	○	○	○			○			○		
1982 (HEP) Multithreading	○			○	○	○	○	○				○	
1982 (CMU) Systolic Array	○	○	○										○

History of high-speed digital computers

(2) Parallelization in node (Single memory space/OS)

Multiprocessor (SMP: ~8)

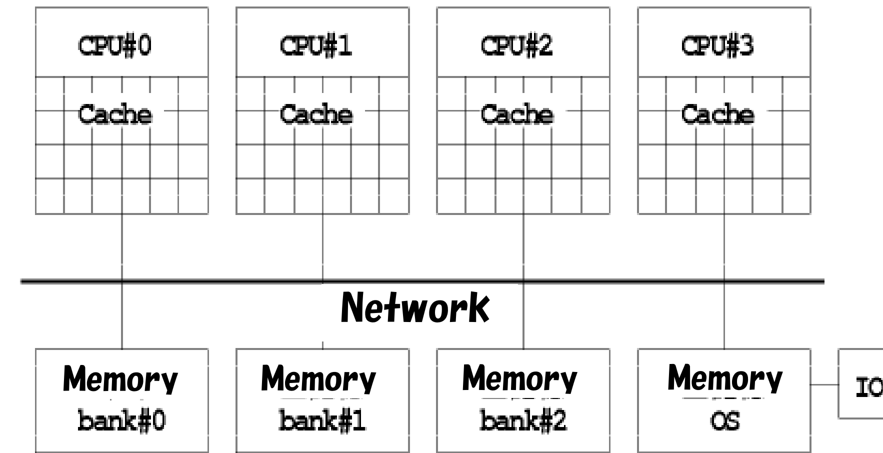
⇒ 1990-Microprocessors

Multiprocessor (NUMA: 128~)

⇒ 2000-for servers

Multicore (~22cores)

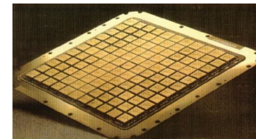
⇒ 2010-for consumers

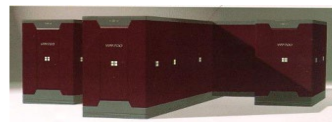


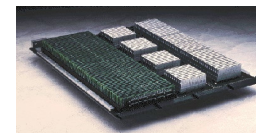
(3) Parallelization in system (Multiple memory space)

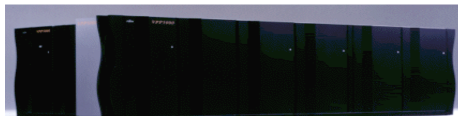
Distributed system (message)

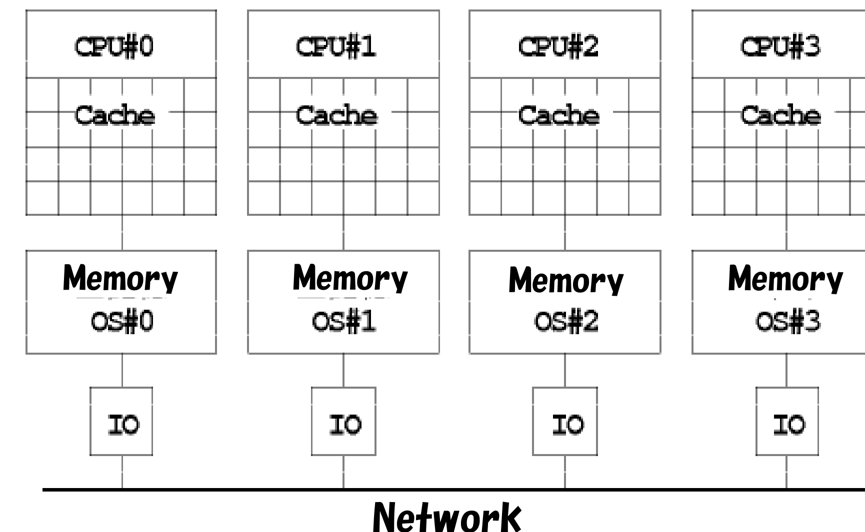
Large scale system











From instruction fetch to storing result

Systematic management of ALU, Register and Memory

Typical functional decomposition

- 1． Fetch

  Fetch an instruction from PC addr. of main memory
- 2． Decode

  Decode the instruction and read data from register

  Prepare control signals for following stages
- 3． Execute

  Execute arithmetic, logical, shift or other operation

  For load/store instruction, calculate effective address (add or sub)
- 4． Memory

  For load instruction, read data from main memory to register

  For store instruction, write data from register to main memory

  Other instruction, do nothing
- 5． Write

  Store result of execution or load data into register

NAIST.

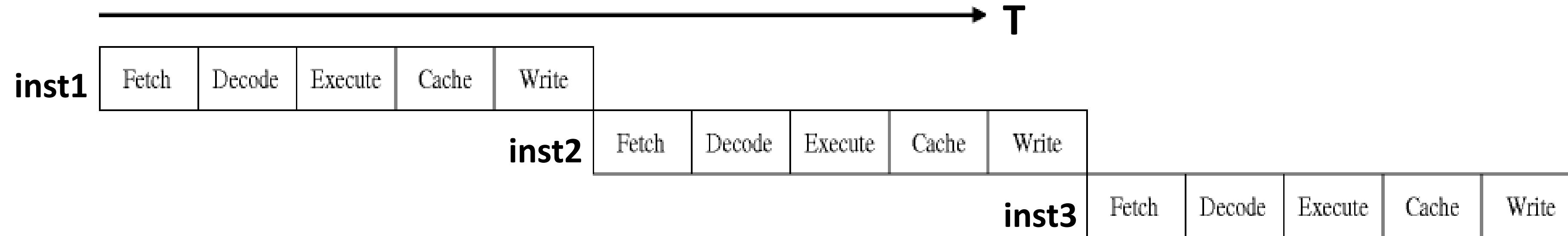
Check operation flow and time

- 1． Fetch
  - Supplying addr. to instruction cache to fetch an instruction
    - Read latency of instruction cache
- 2． Decode
  - Extracting register numbers form the instruction
  - Then supplying them to register file to get resister contents
    - Read latency of register file
- 3． Execute
  - Supplying input value to getting output through LogN step gates
    - Latency of LogN step gates
- 4． Cache
  - For load instruction, supplying addr. to getting data
  - For store instruction, supplying addr. to writing data
    - Read/write latency of data cache
- 5． Write
  - Supplying register number to storing result of execution or load
    - Write latency of register file

NAIST

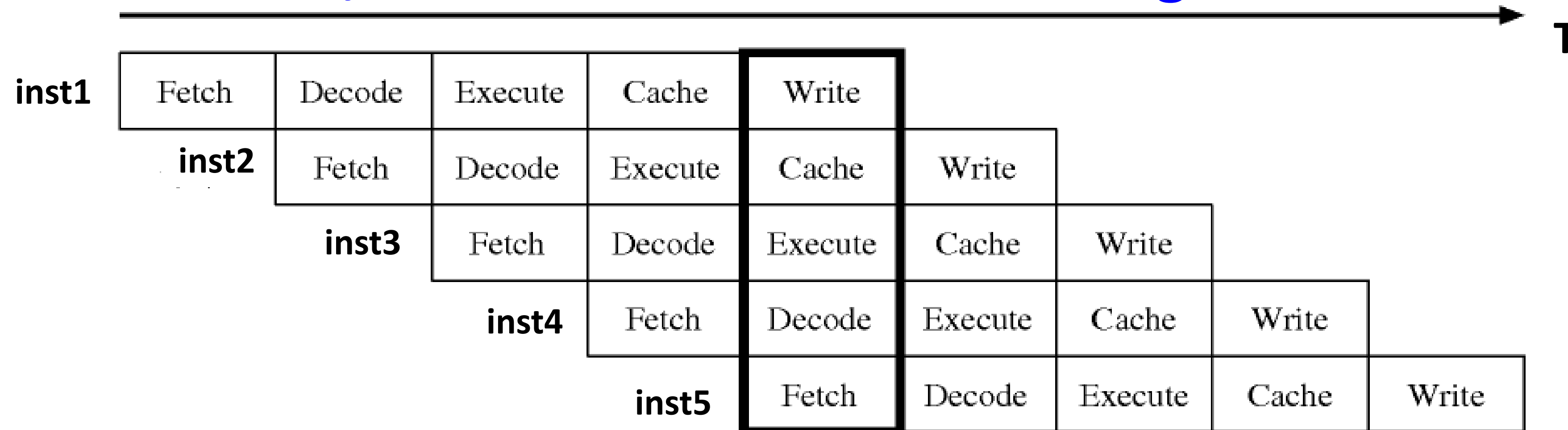
Pipeline execution from fetch to write

Just do it sequentially … like ancient 8bit CPU (no cache)



Fast execution by stage parallel execution

- Since temporal storage (pipeline registers) is required, Improvement is smaller than the number of stages
- Even slower, if execution time of each stage is not balanced



Cache latency is a guideline of "stage latency"

- **Level 1 cache (small and fast)**
  Ex. 64byte width x 256 (16K byte)
  For simple CPU, cache speed roughly determines total performance
- **Register file (smaller and fast)**
  Ex. 32bit width x 32 (128 byte)
  Fast enough but double speed operation is required (discuss later)
- **LogN stage logic gates**
  Ex. 32bit adder (thousands of transistors)
  Fast enough but write latency is not negligible due to its long path
  (discuss 3rd day)

Other basic components (Longer latency than pipeline stage latency)

- Level 2 cache (middle capacity and middle speed)
  > 2MB
- Main memory (large but small)
  > 2GB

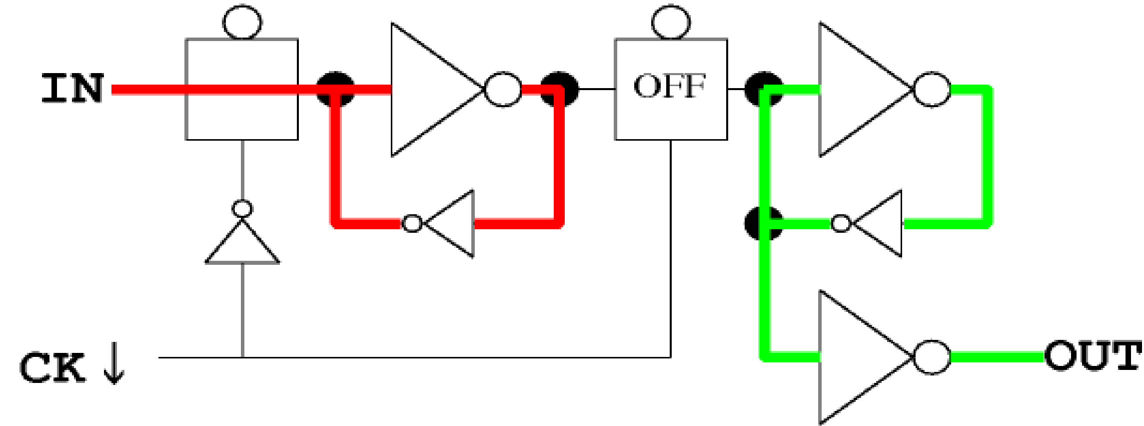
Pipeline Register

Manage signal propagation

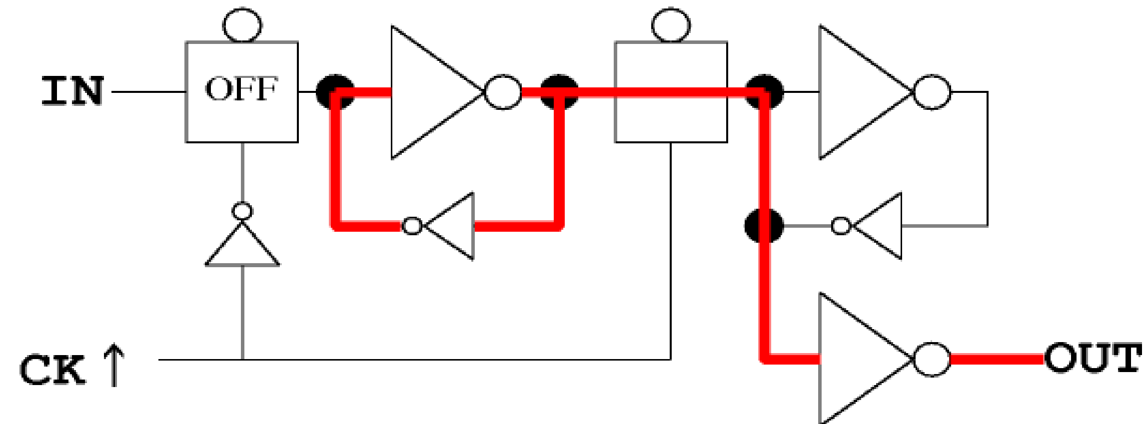
NAIST

Mechanism of master slave latch

When CK=0, input value is propagated to 1st stage, output is unchanged



When CK=1, input value is propagated to output



NAIST

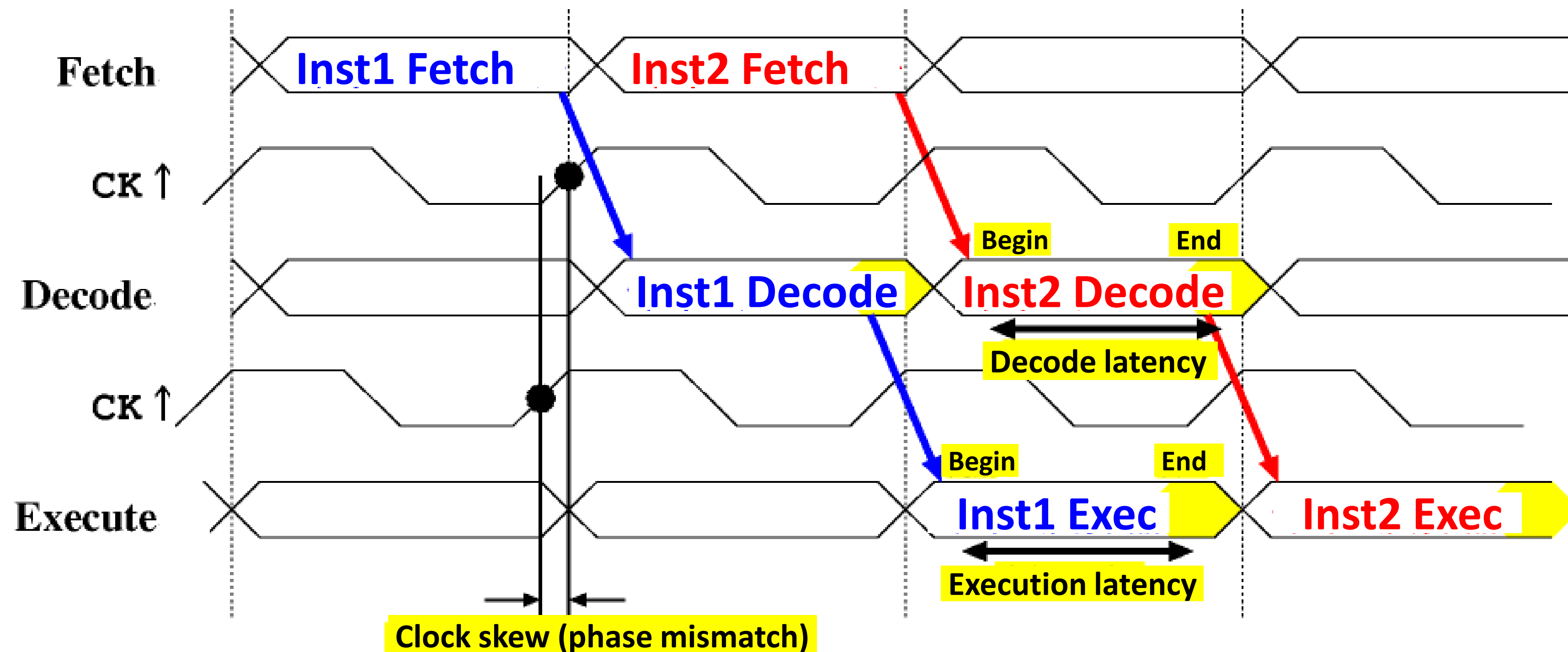
Operation of master slave latch

When CK↑, each pipeline stage starts operation

- **If result of previous stage is arrived, propagation is blocked by latch**
- **If result is not arrived when CK↑, propagation is failed**

  Therefore, to supply higher frequency clock without latency improvement causes malfunction

  Clock skew is also common reason of malfunction

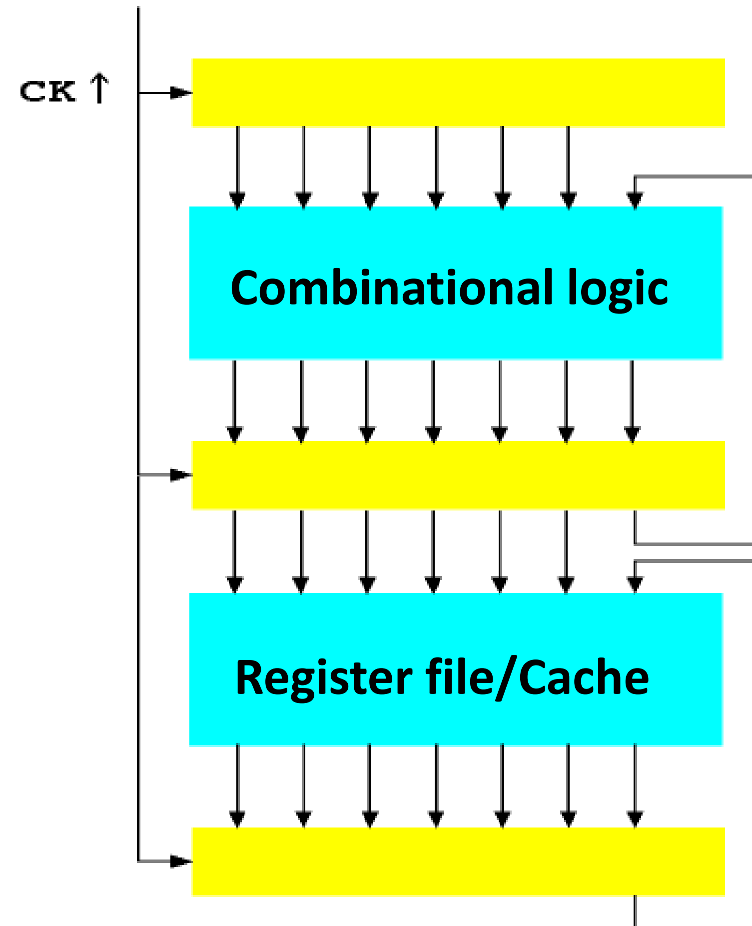


NAIST

Locate pipeline registers between stages

Clock and pipeline register

- By connecting output to input of previous stage, information can be hold
  Ex. Use +4 logic as combinational logic to make program counter



Recall the previous class #2001

Source code

```
int R1[100];          /* reg r1: top address of array R1[] */
int R2;               /* R2: index for array R1[] */
R1[1] = R1[1]+8;      /* add 8 to second element of R1[] */
R1[R2]= R1[1]-R1[R2]; /* reg r2:  R2 */
```

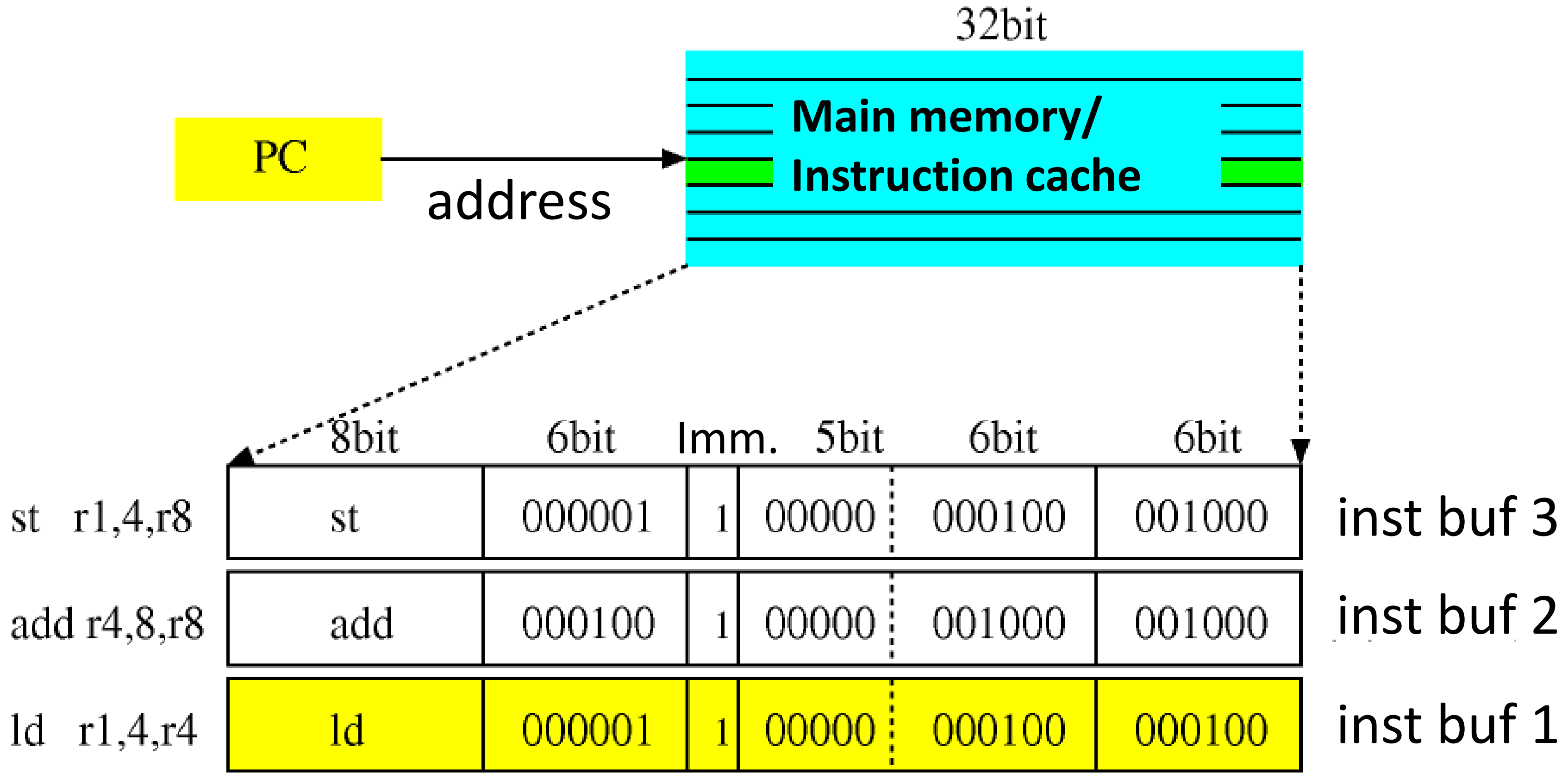
Instructions

```
ld  r1,4,r4       r4 ← mem[r1+4]
add r4,8,r8       r8 ← r4 + 8
st  r1,4,r8       Mem[r1+4] ← r8
ld  r1,r2<<2,r5   r5 ← mem[r1+r2*4]
sub r8,r5,r9      r9 ← r8 – r5
st  r1,r2<<2,r9   Mem[r1+r2*4] ← r9
```

Computer repeats Decode, Read, Execution and Write-back.

Fetch – Fetch an instruction

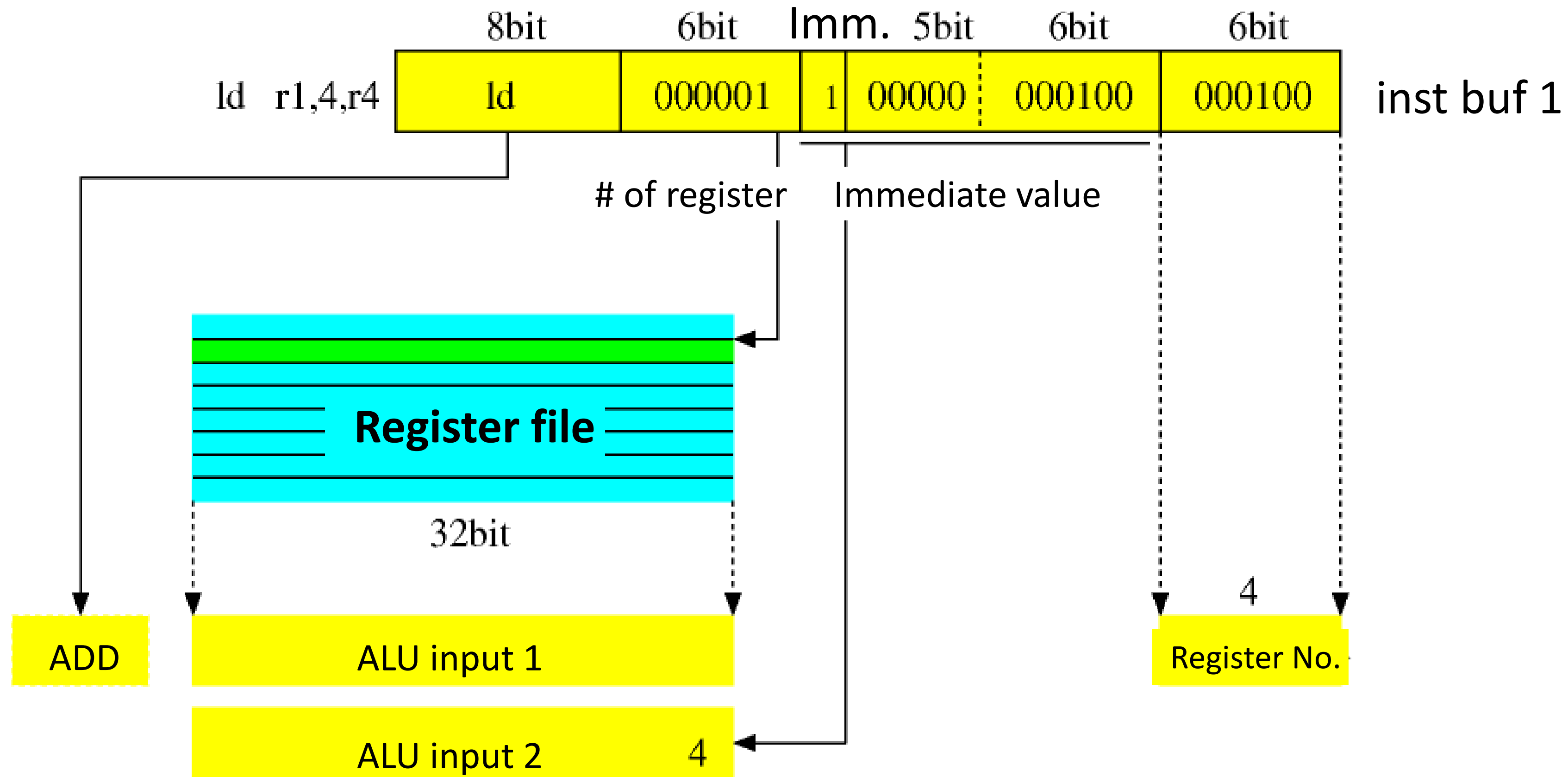
Read PC addr. of Main memory and write result to instruction buffer



NAIST

Decode – Decode instruction

Decode instruction then prepare control signals for ALU, input data, output register number

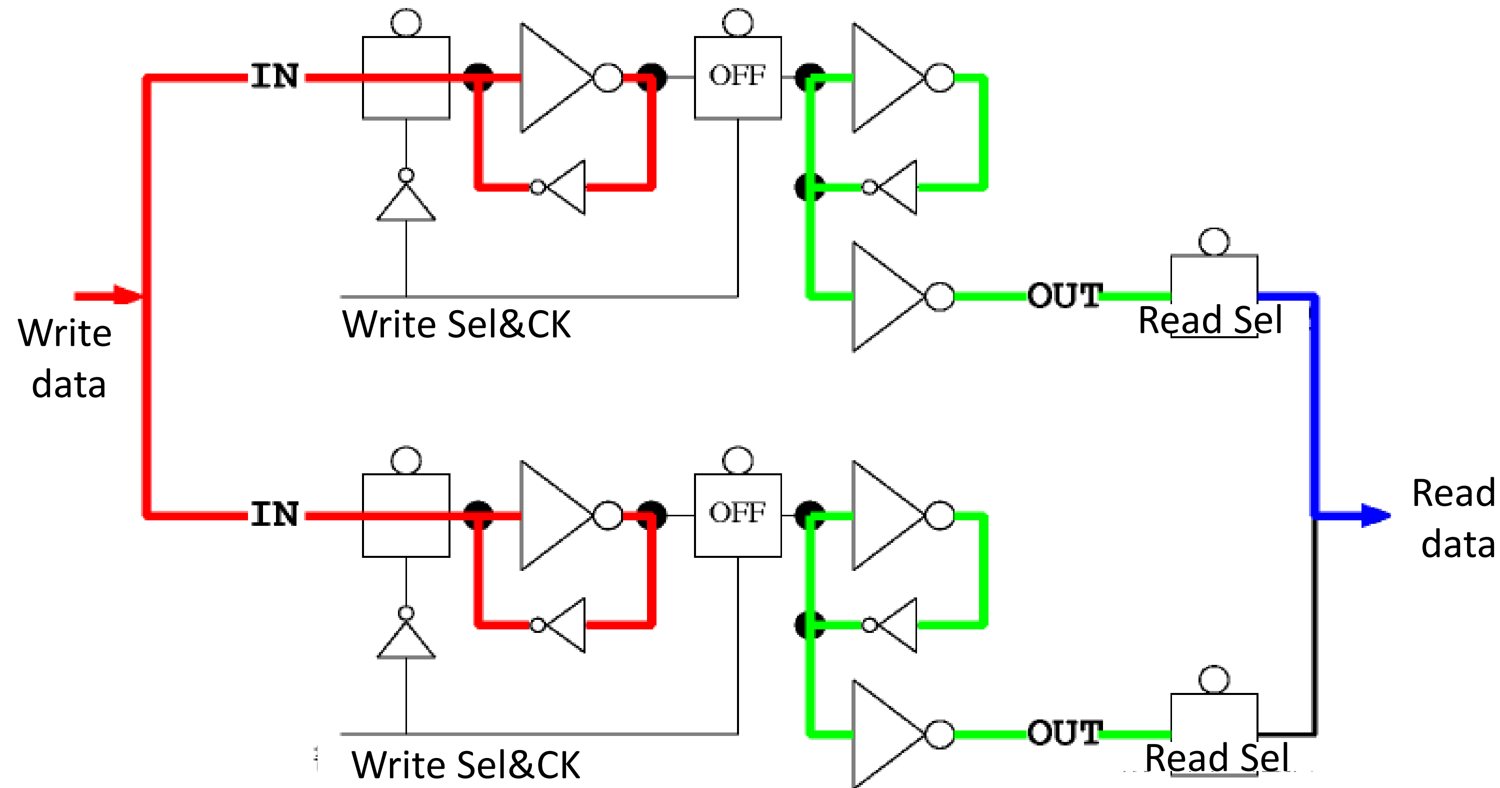


NAIST

Basic structure of register file

Select one of N registers and Write or Read it

- **For write, decode logN bit then set one of N WriteSel&CK to 0**
- **For read, decode logN bit then set one of N Read sel to 1**

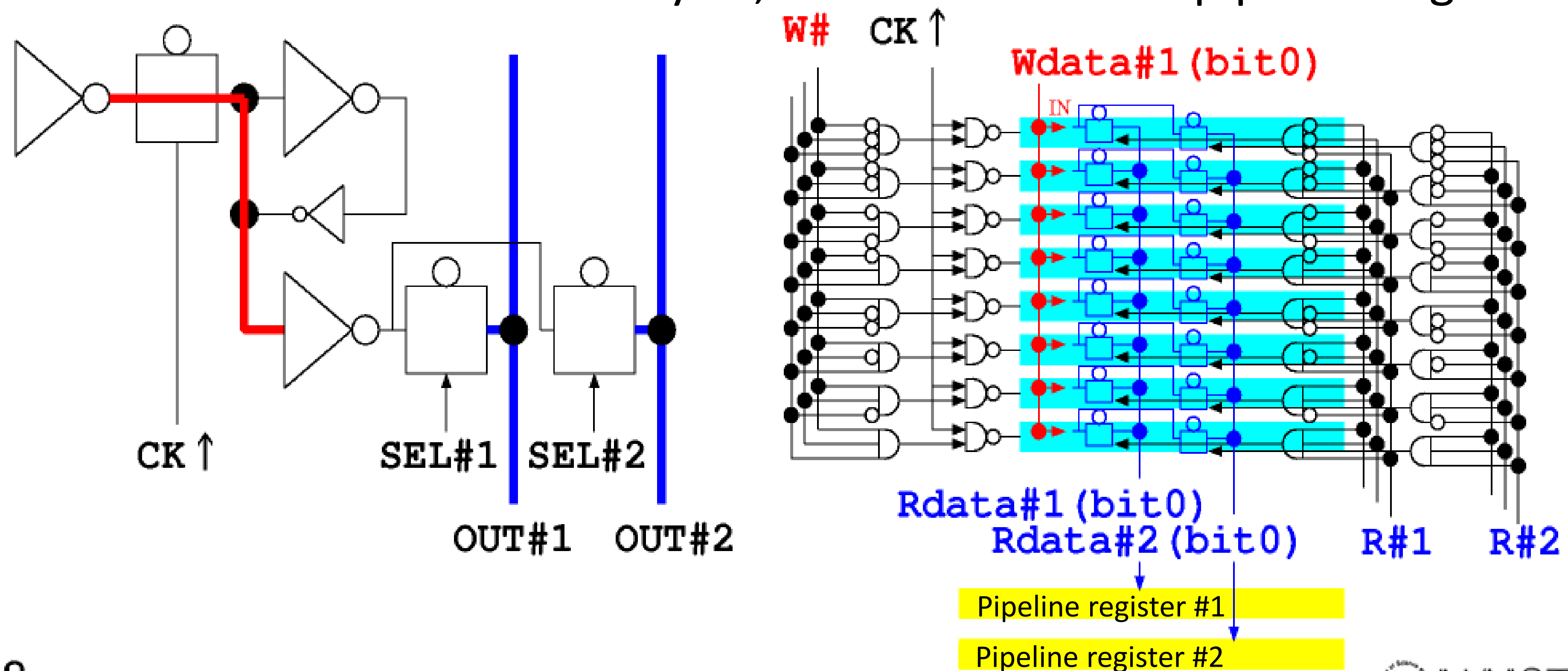


Overall structure of register file

For every cycle execution, following function is required for register file

- **Read from either same or different two registers (two read port)**
- **Write to one register (one write port)**

At the end of first half cycle, Wdata is available at Rdata
At the end of second half cycle, Rdata is written in pipeline register



NAIST

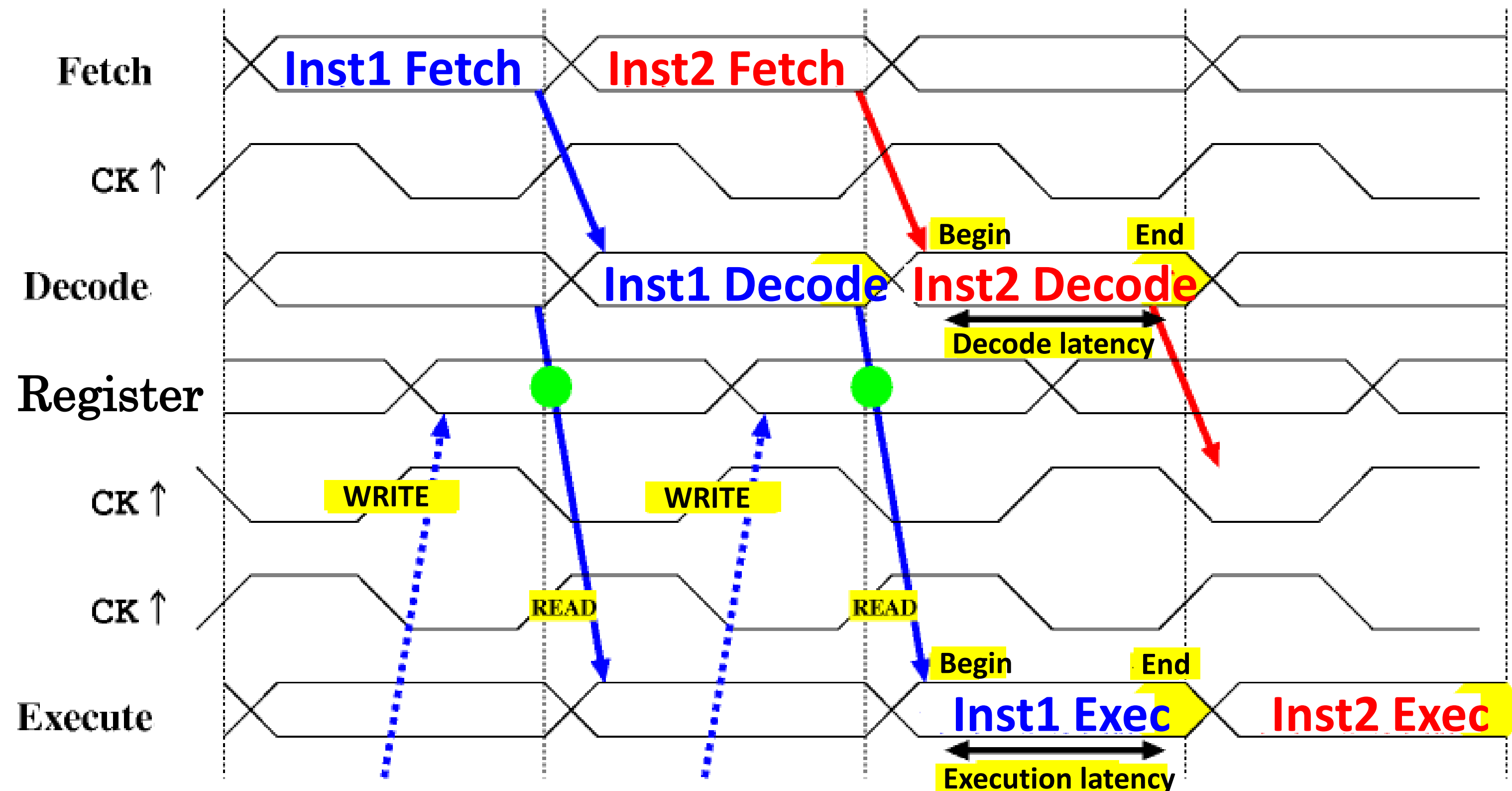
Antiphase operation of pipeline register and register file

If register file is composed of latch, half cycle READ/WRITE are possible

- Write in first half cycle and read in second half cycle
- Register bypassing (explain later) can be omitted

If register file is composed of memory, half cycle operations are not possible

- Register bypassing (explain later) is required

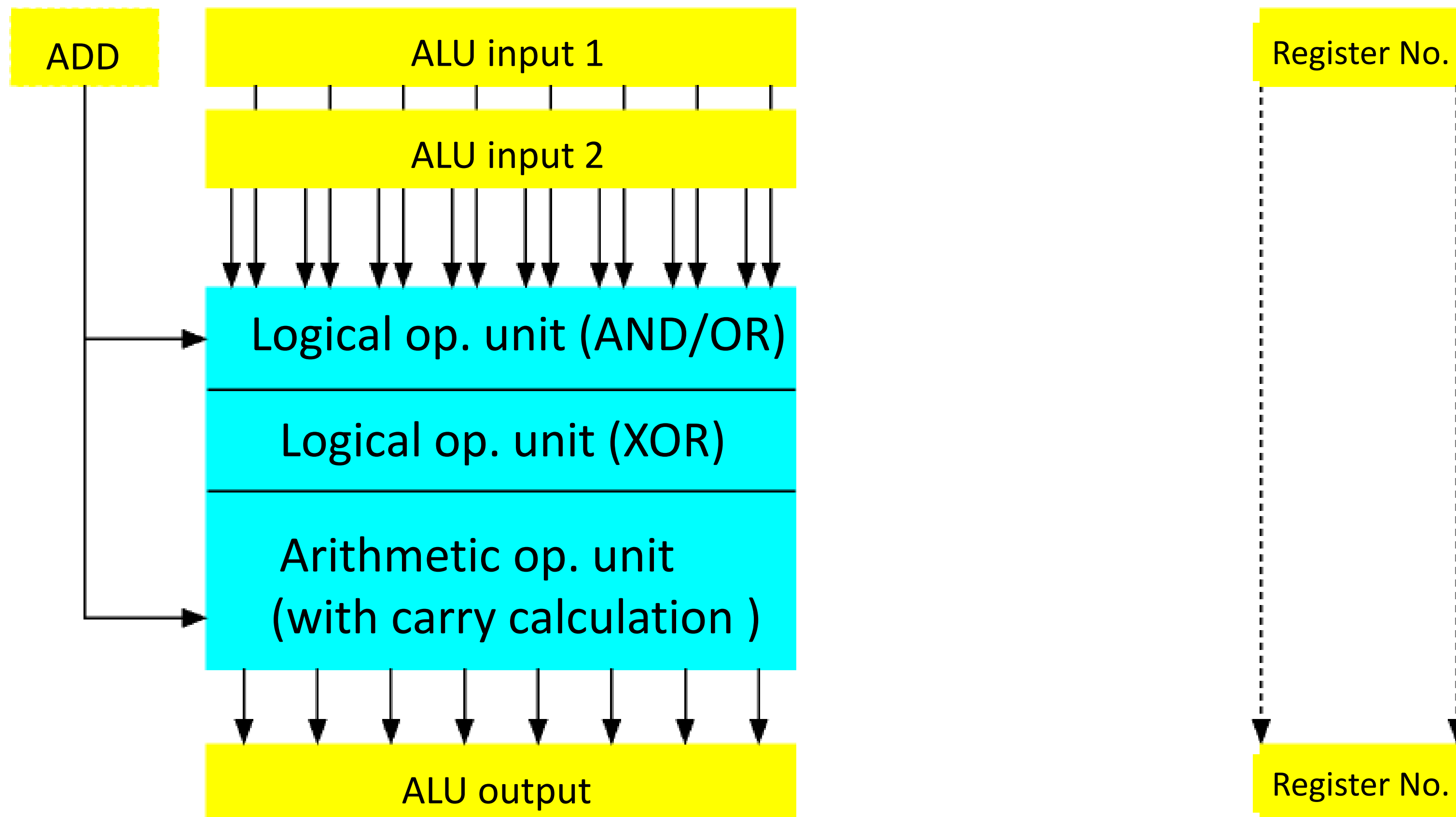


NAIST

Execute

Do operation that is specified by control signal, result is stored temporally

- **Do address calculation for load/store instructions**

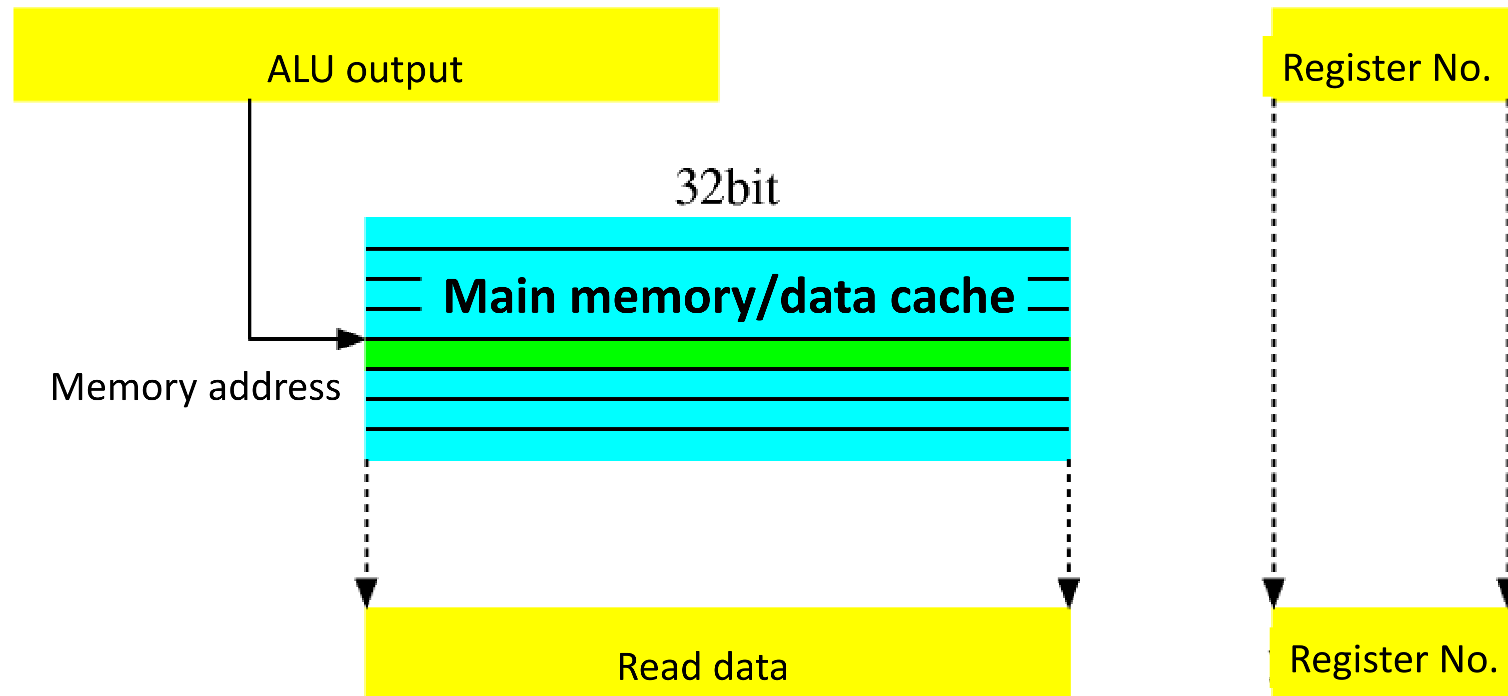


NAIST

Cache – Read operand

Access memory with calculated address and store result temporally

- **If virtual address cache, TLB is in this stage**
- **If store buffer is implemented, it is also in this stage**

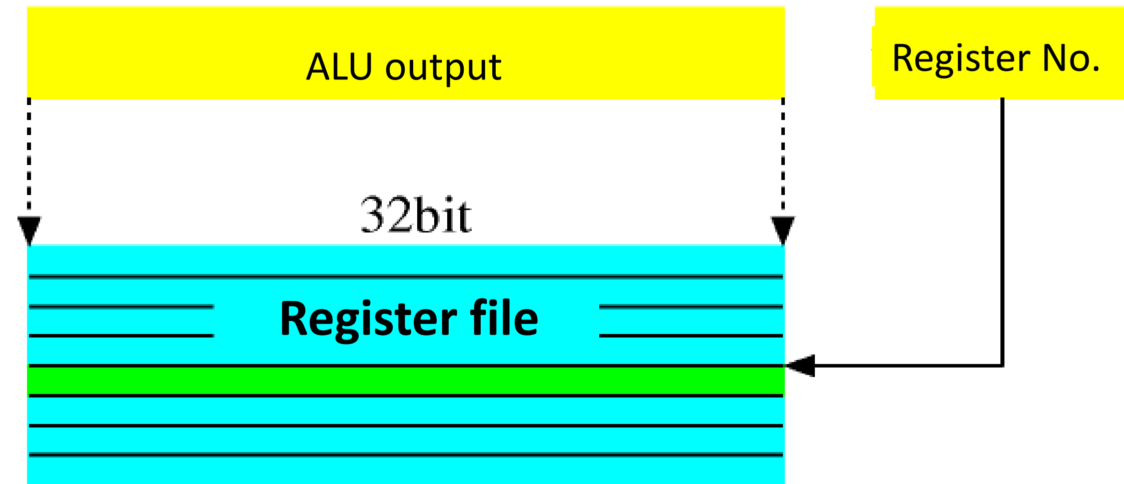


NAIST

Write – Write result

Write result to register file

- Note that there is register file in Decode stage
- If it is composed of latch, half cycle READ/WRITE is possible
- If it is composed of memory, half cycle operation is not possible

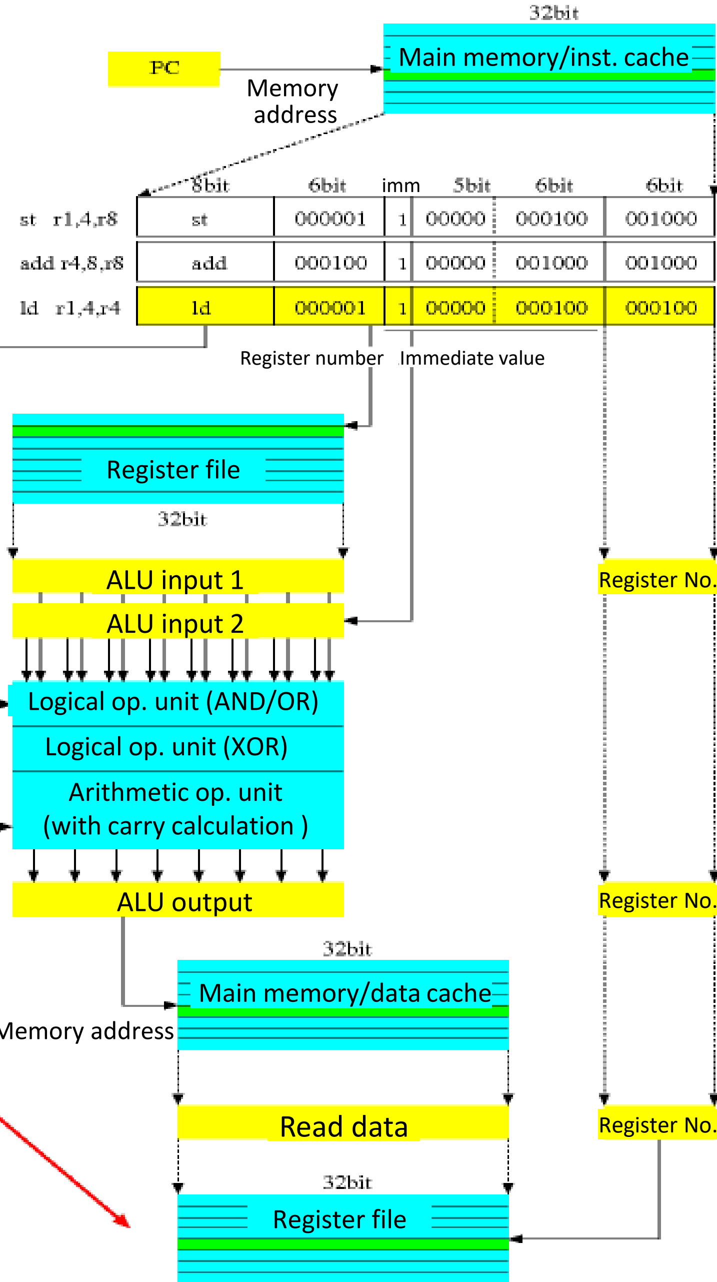


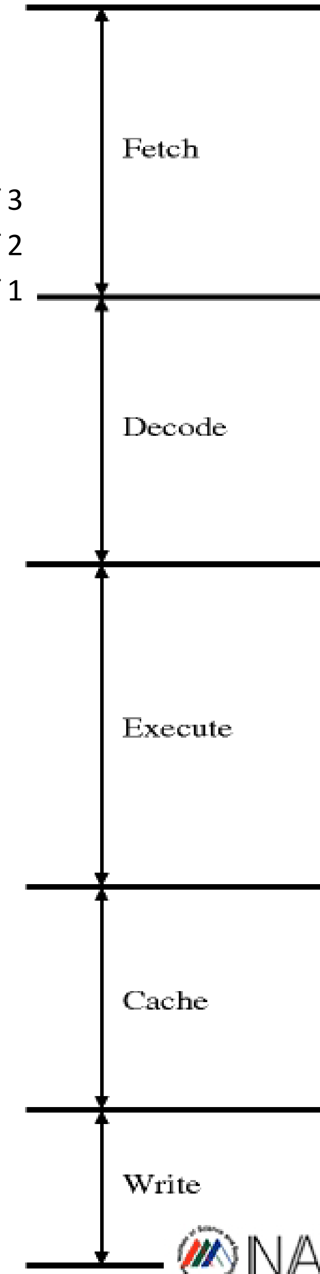
NAIST.

Simply connect them together

Temporally register

Long latency structure

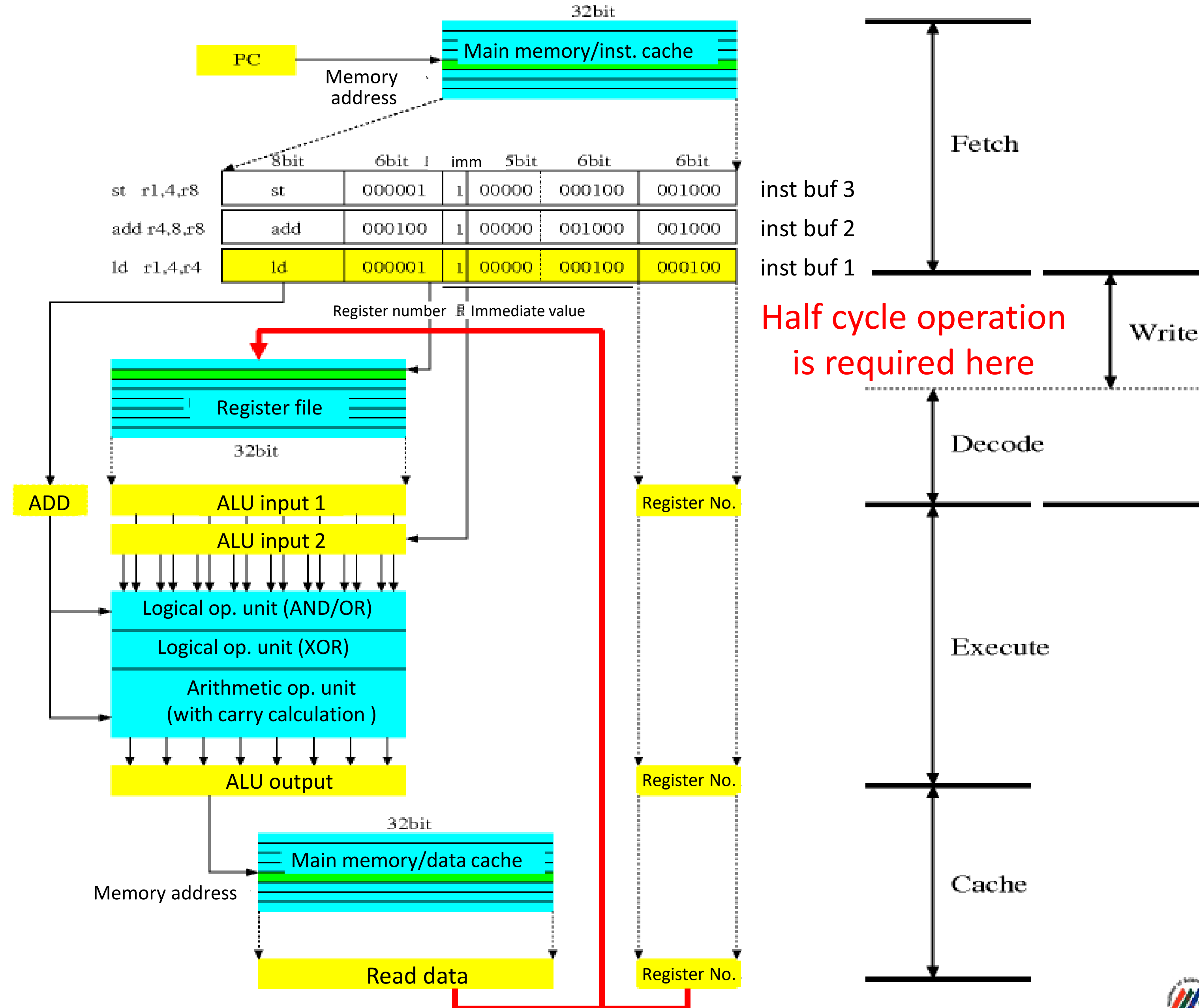




Why there are two register files? Impossible!

NAIST

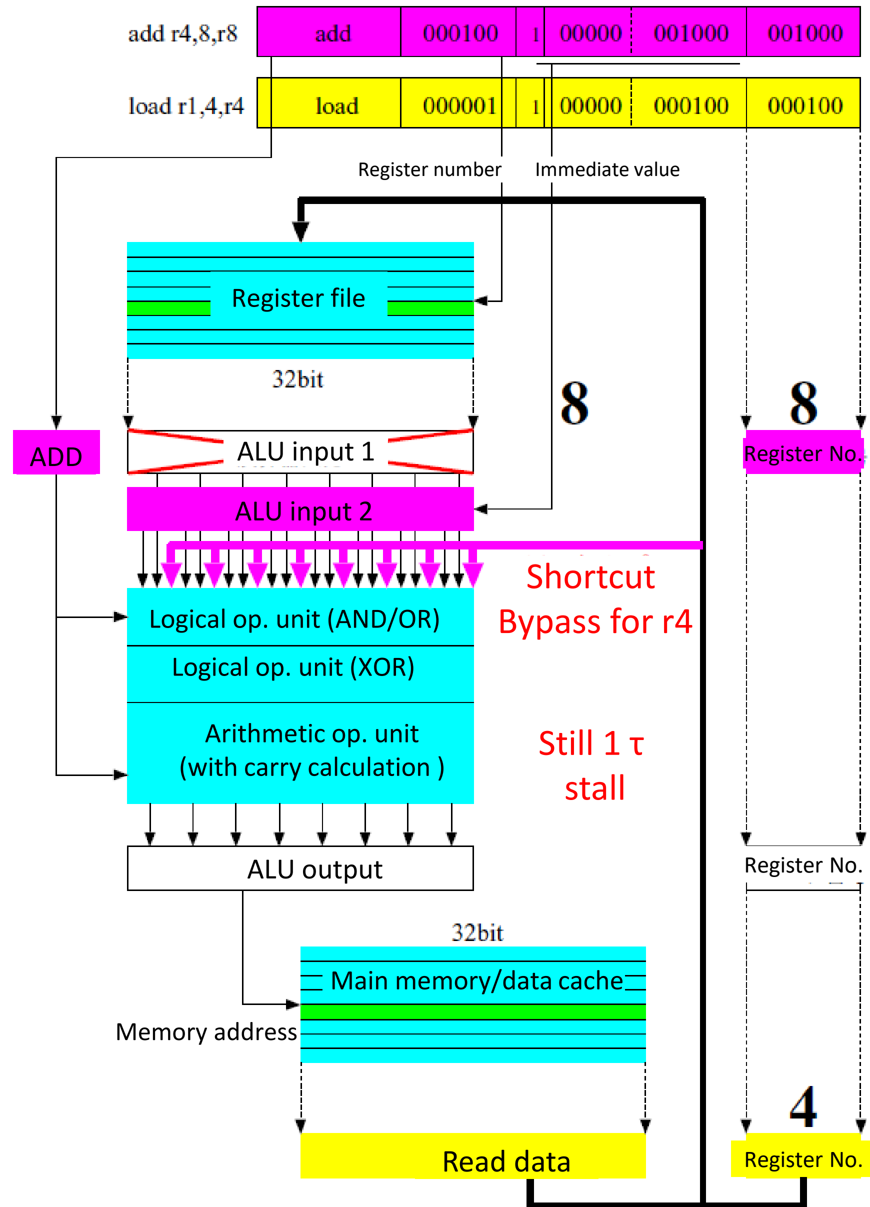
Connect them correctly (Decode and Write are in one cycle)



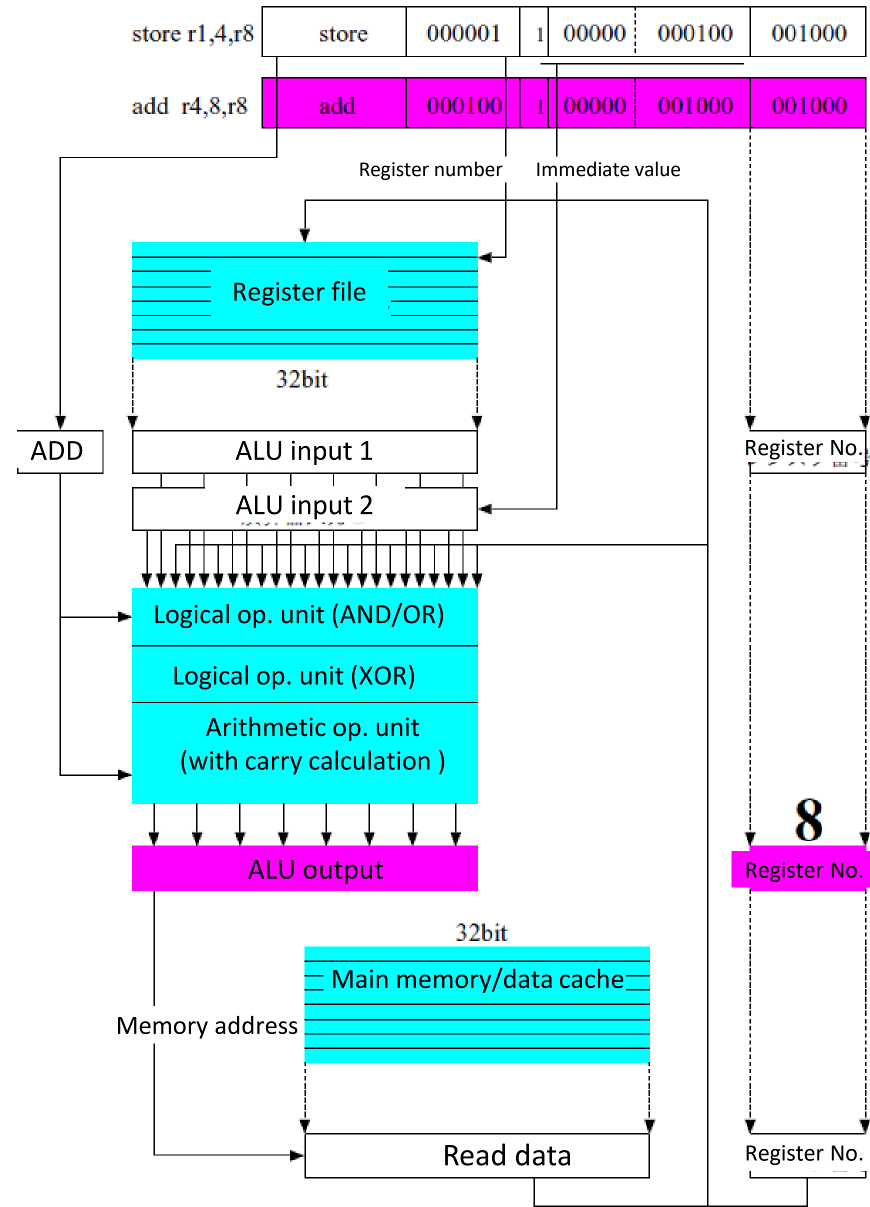
Half cycle operation is required here

NAIST

Load then add (ld-use penalty if dependency is exists)

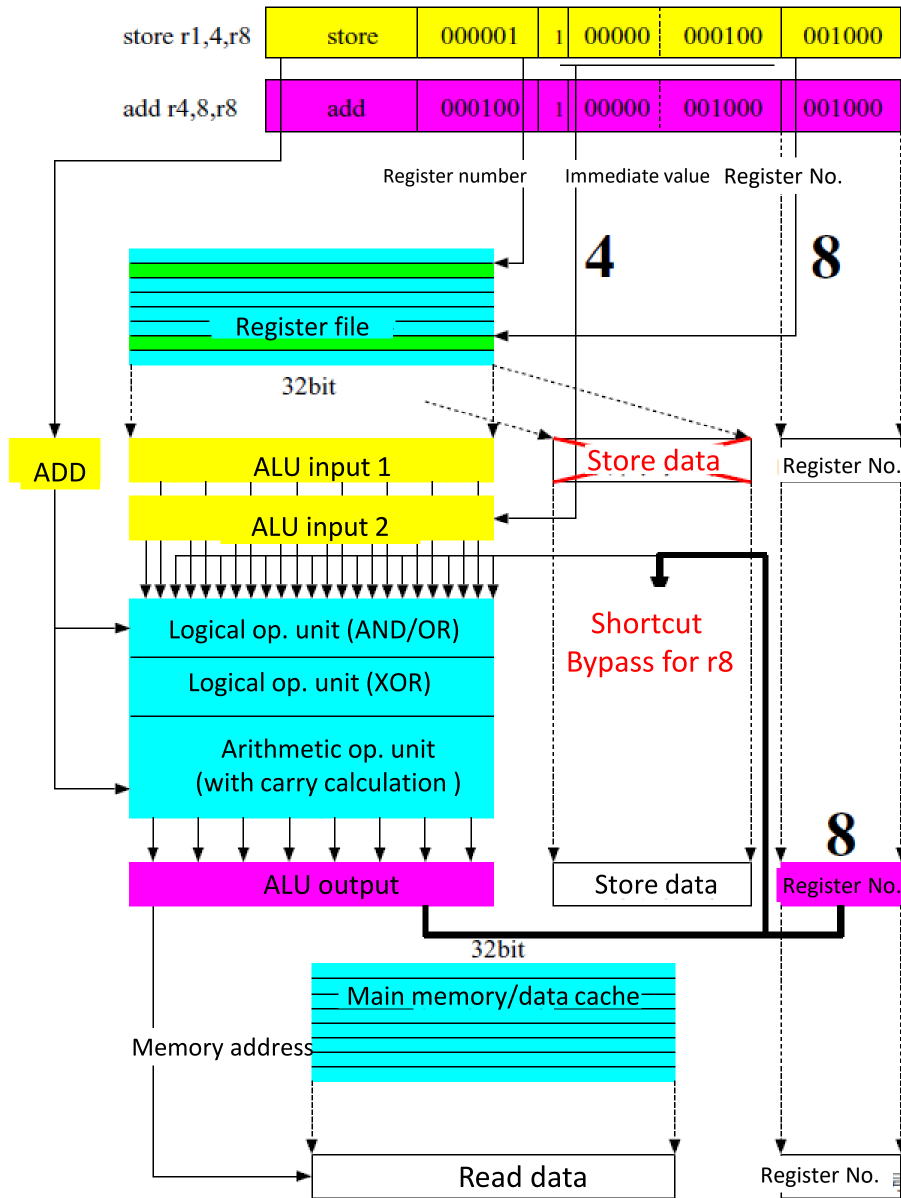


When load is done

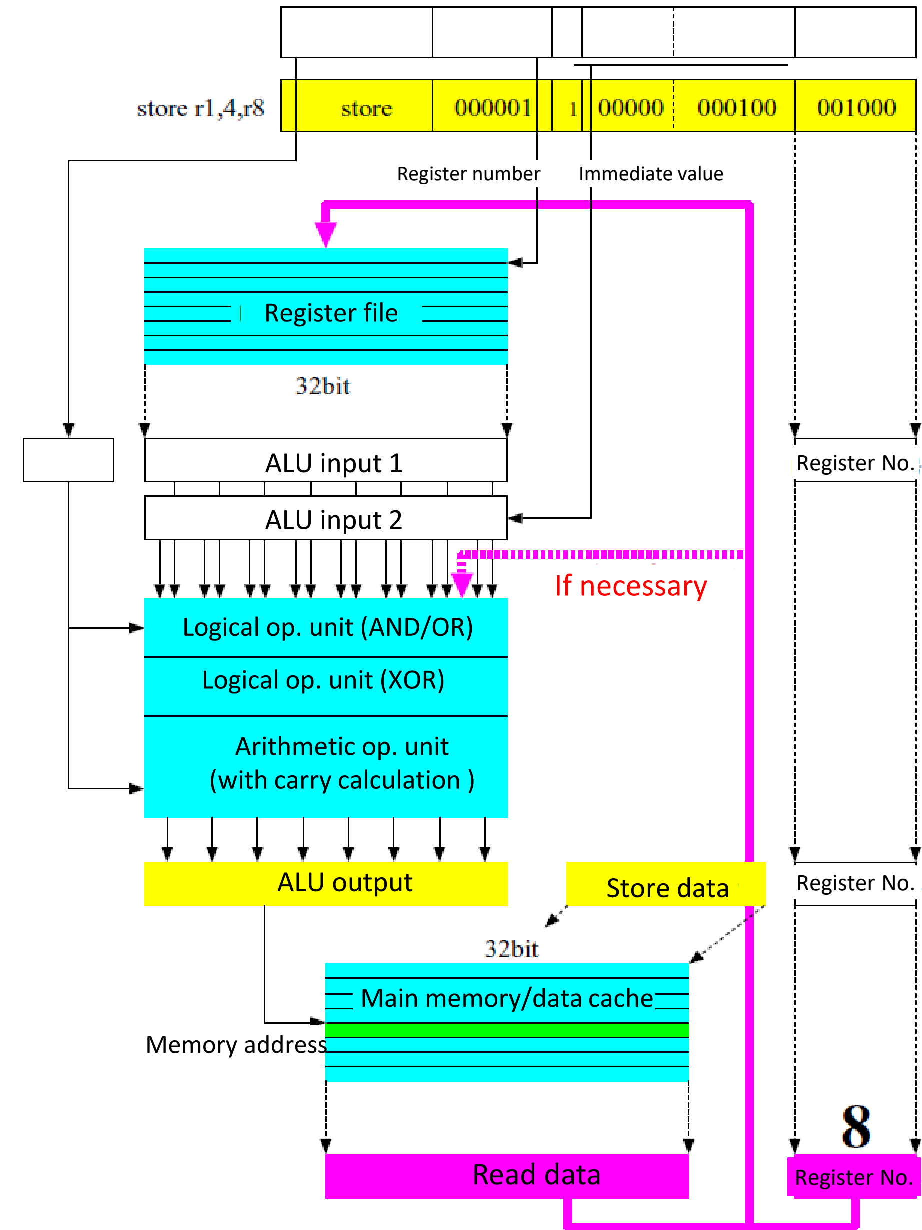


When execution of add is done

Add then Store (no penalty by bypassing)



When execution of add is done



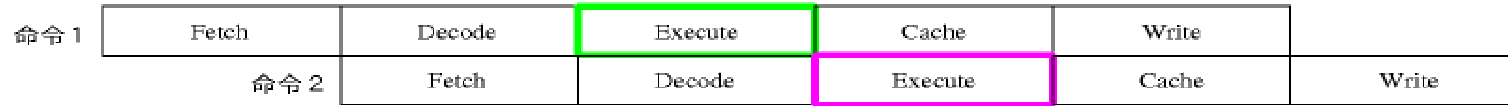
When add is completed

Superscalars

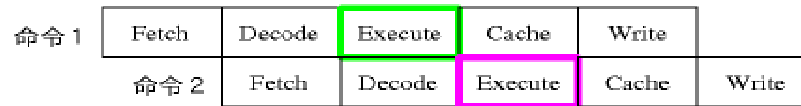
for gathering instructions to be executed simultaneously

Inefficiency of fake high-frequency

- **Deep pipeline stages requires many instructions**

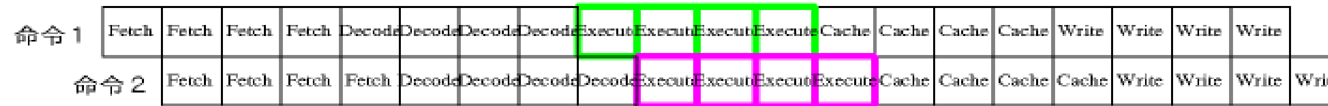


High-speed device for true speed-up (x2 frequency)



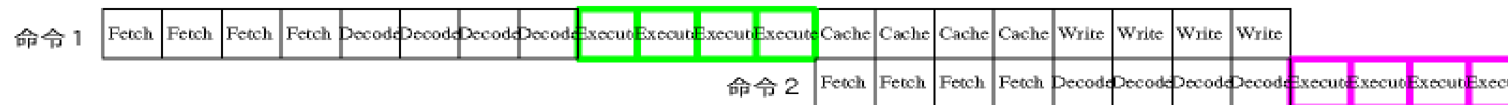
Super pipelining for fake speed-up (x2 frequency)

Fake speed-up (x4 frequency)



Dependent instruction is delayed (same speed)

Branch instruction is also delayed (very slow)



Re-ifetch due to branch miss-prediction

NAIST

Importance of gathering instructions

High-frequency is inefficient. (Power ∝ F^3)

- **Employing parallelism is most important**
- **First superscalar executes neighbor two instructions**
- **Modern superscalars gather from large window.**

命令 1	Fetch	Fetch	Fetch	Fetch	Decode	Decode	Decode	Decode	Execute	Execute	Execute	Execute	Cache	Cache	Cache	Cache	Write	Write	Write	Write
命令 2	Fetch	Fetch	Fetch	Fetch	Decode	Decode	Decode	Decode	Execute	Execute	Execute	Execute	Cache	Cache	Cache	Cache	Write	Write	Write	Write

Accurate branch prediction is required

命令 1	Fetch	Fetch	Fetch	Fetch	Decode	Decode	Decode	Decode	Execute	Execute	Execute	Execute	Cache	Cache	Cache	Cache	Write	Write	Write	Write				
命令 2													Fetch	Fetch	Fetch	Fetch	Decode	Decode	Decode	Decode	Execute	Execute	Execute	Execute

Re-ifetch due to branch miss-prediction

NAIST

Obstacles for simultaneous execution of instructions

Three types of dependency in programs

- **Flow dependency (write ⇒ read)**
  Succeeding insn refers the result of preceding insn

```
add r1,r2 → r8
sub r8,r3 → r4
```

Fundamental data dependency is hard to resolve

- **Anti-dependency (read ⇒ write)**
  Succeeding insn should wait for preceding read

```
sub r8,r3 → r4
add r5,r6 → r8
```

Register renaming can remove dependency

- **Output dependency (write⇒ write)**
  Succeeding insn should wait for preceding write

```
add r1,r2 → r8
sub r8,r3 → r4
add r5,r6 → r8
sub r8,r3 → r7
```

Register renaming can remove dependency

Basic idea to remove dependency

- **The usage of registers are defined by ABI (application binary interface), then compilers don't have enough registers.**
- **Register number is just representing data dependency. No need to assign physical register as specified.**
- **If a new destination register is assigned to every insn, anti-dependency and output dependency are eliminated.**
  - **Specified register number: Architectural register**
  - **Real register number: Physical register**

NAIST

Register renaming

Previous example: no parallelism

```
add  r1,r2 → r8
sub  r8,r3 → r4
add  r5,r6 → r8
sub  r8,r3 → r7
```

Renamed destination registers

```
add  r1,r2 → p0(r8)
sub  r8,r3 → p1(r4)
add  r5,r6 → p2(r8)
sub  r8,r3 → p3(r7)
```

Remapped source registers

```
add  r1    ,r2 → p0
sub  p0(r8),r3 → p1
add  r5    ,r6 → p2
sub  p2(r8),r3 → p3
```

2-instructions/cycle doubles the performance

```
add r1,r2 → p0 と add r5,r6 → p2
sub p0,r3 → p1 と sub p2,r3 → p3
```

Releasing physical register

When each physical register is released ?

Physical register seems permanently blocked,
because future consumer is unknown.

After correct branch-direction is determined and preceding insns are finished, architecture register is confirmed.

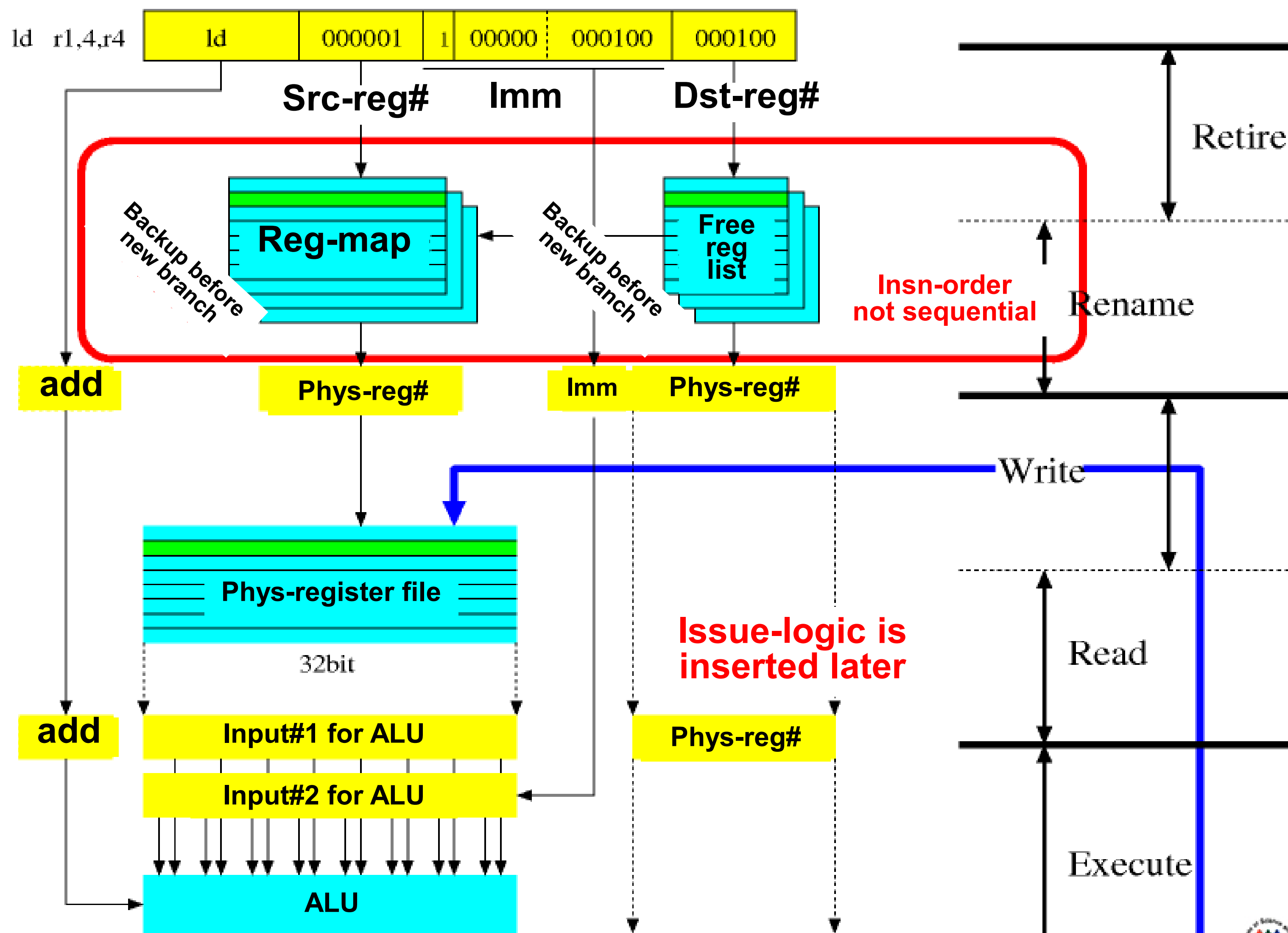
Then corresponding physical register is released.

- Physical register scheme: uses unified register space
  Map for "arch-reg⇒phys-reg" is maintained
- Reorder buffer scheme: uses architectural register space

NAIST

Physical register scheme

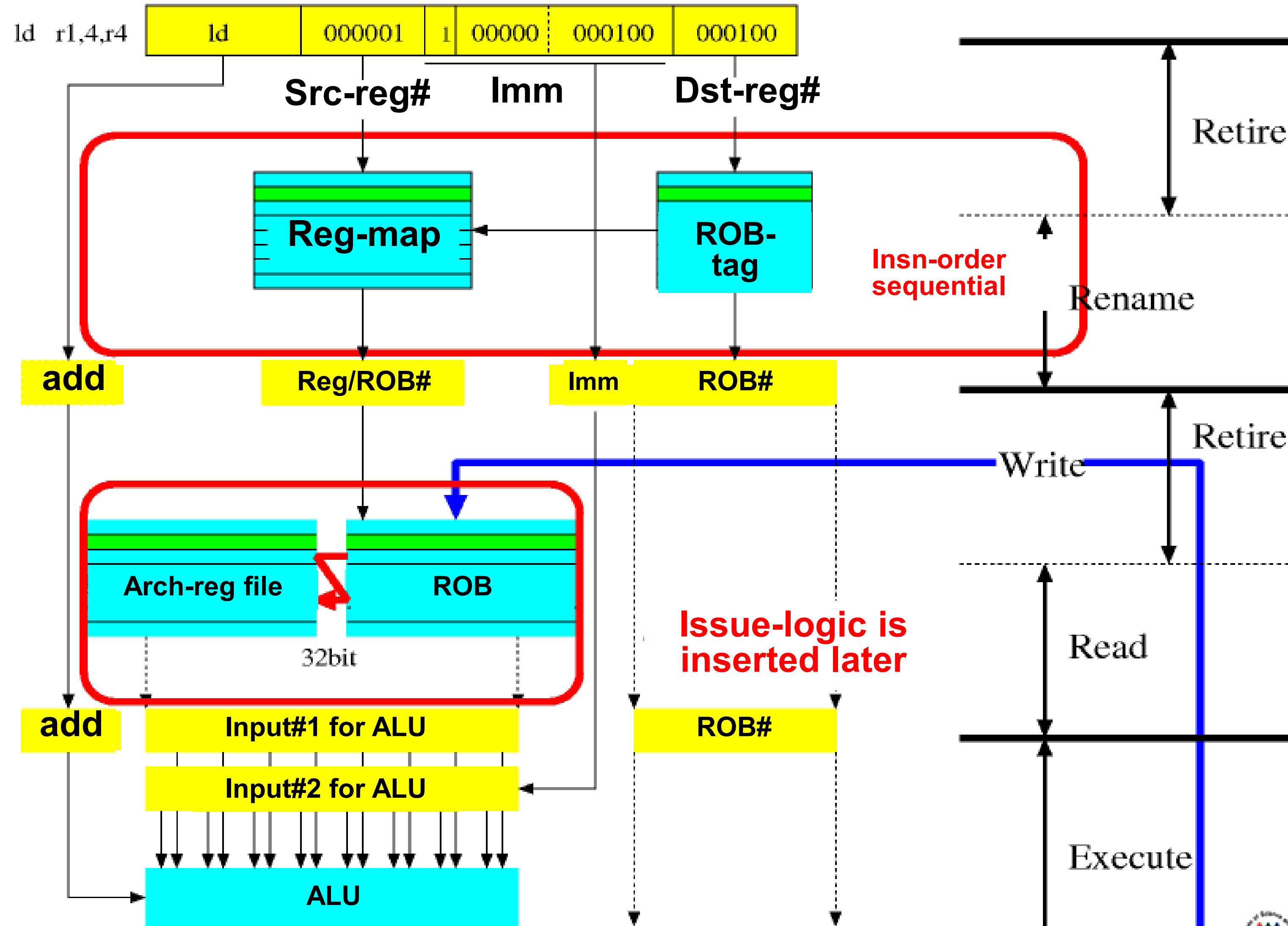
Before new branch, snapshot of reg-map is required.



NAIST

Reorder buffer scheme

Destination register is allocated in reorder buffer.

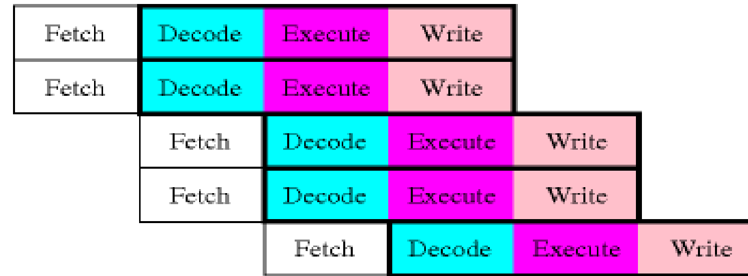


NAIST

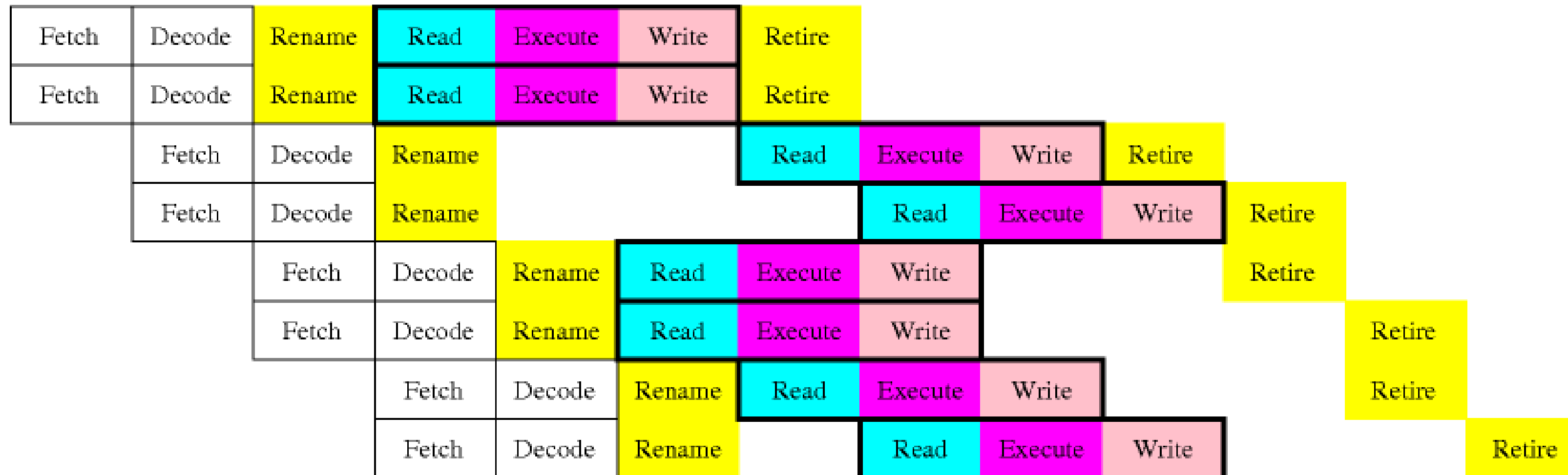
Issue mechanism for executing multiple instructions

Instruction window

It is easy to issue neighbor instructions.
But parallelism is limited.

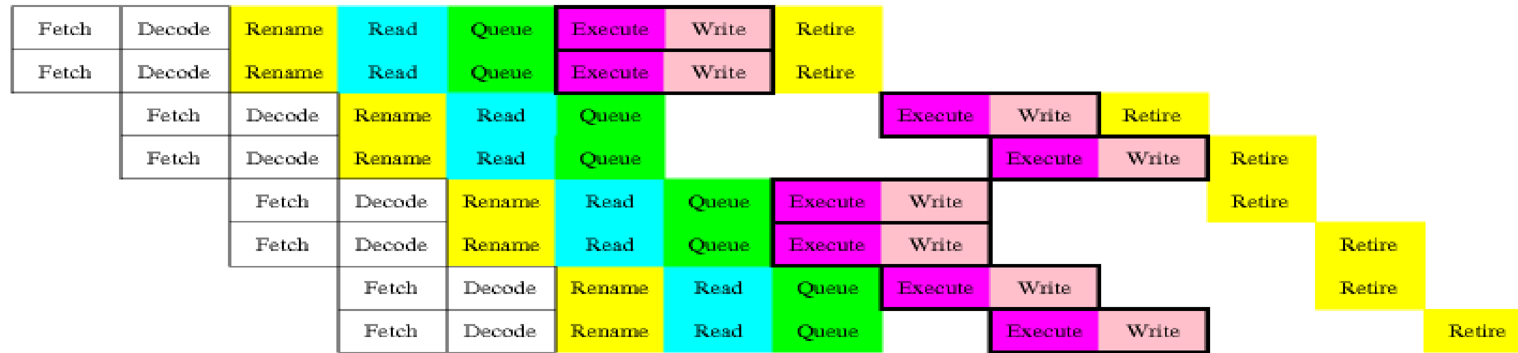


Keep many instructions in rename-retire window, and select as many as possible.

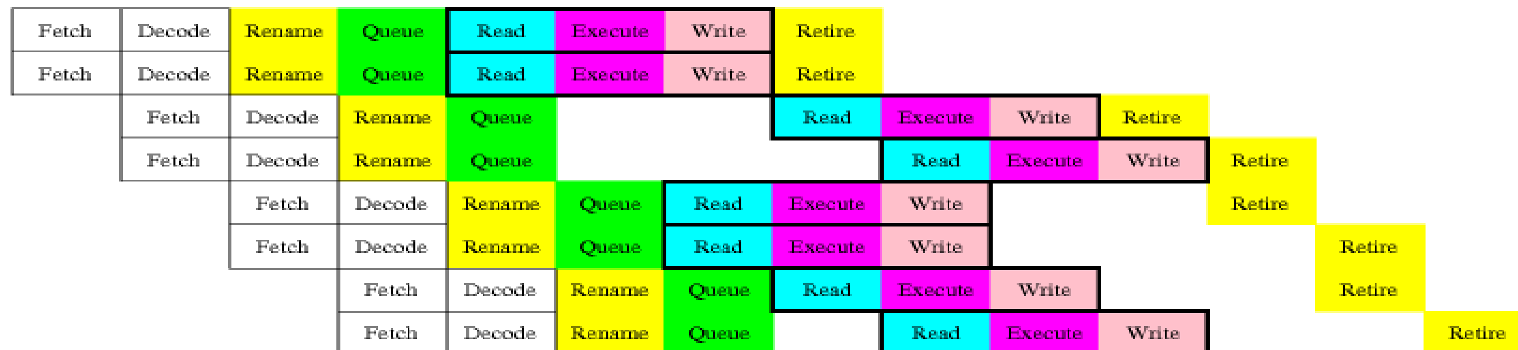


Where instructions are enqueued

- **Reservation station scheme: just before execution stage All operands should be read before enqueued.**



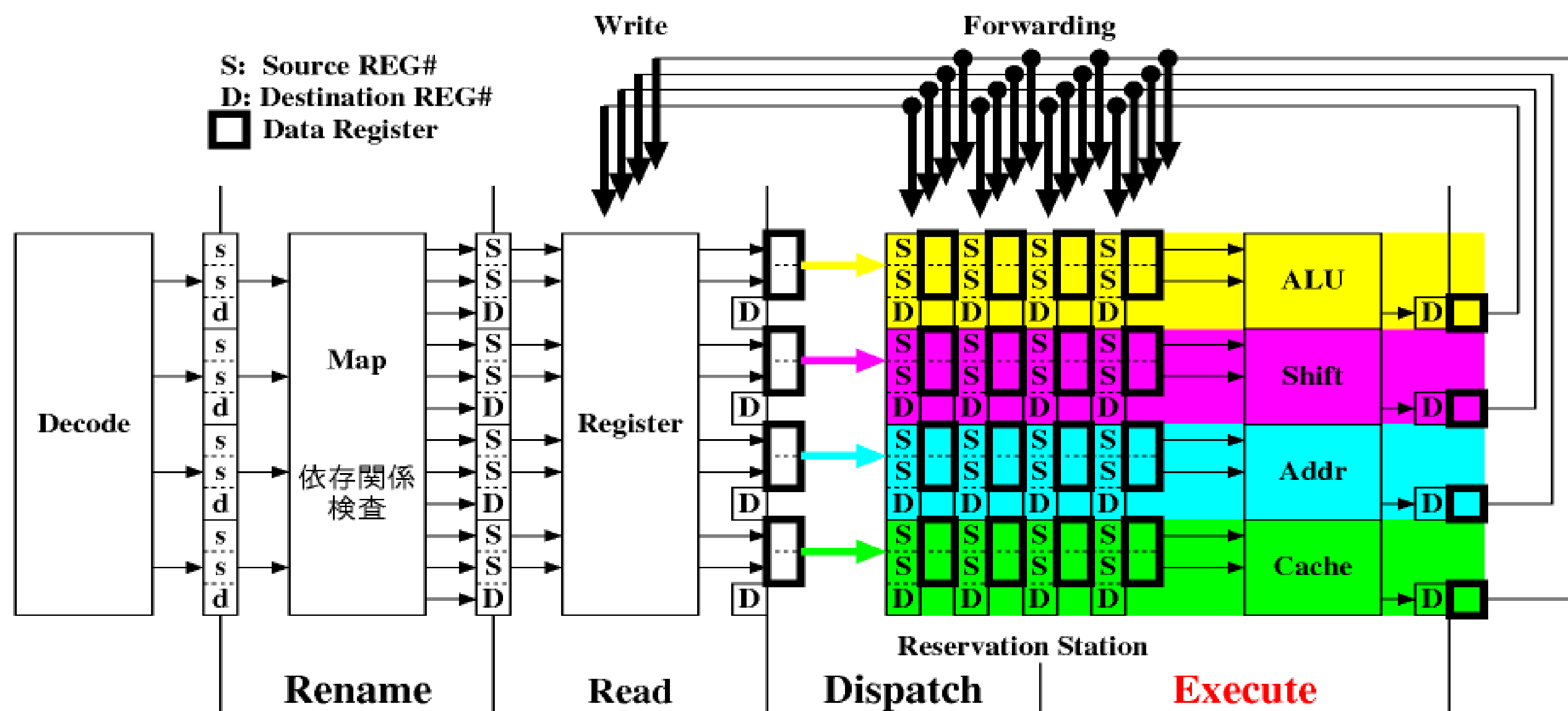
- **Centralized instruction window scheme: just before reg-read stage**



NAIST

Reservation station scheme

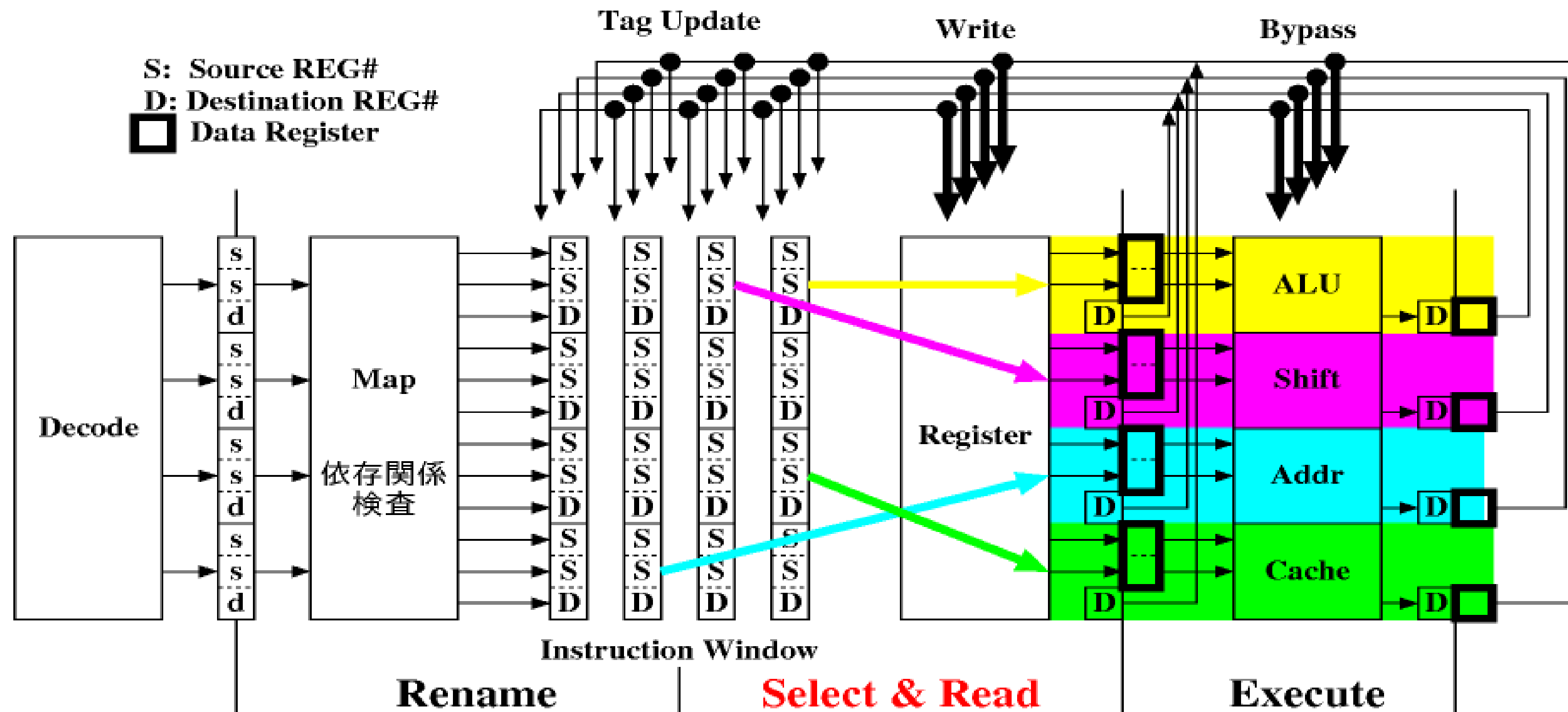
For back-to-back execution, the result should be forwarded to next execution every cycle.



NAIST

Centralized instruction window scheme

After issued, registers are read.
“Select & read” stage is critical.



VLIW

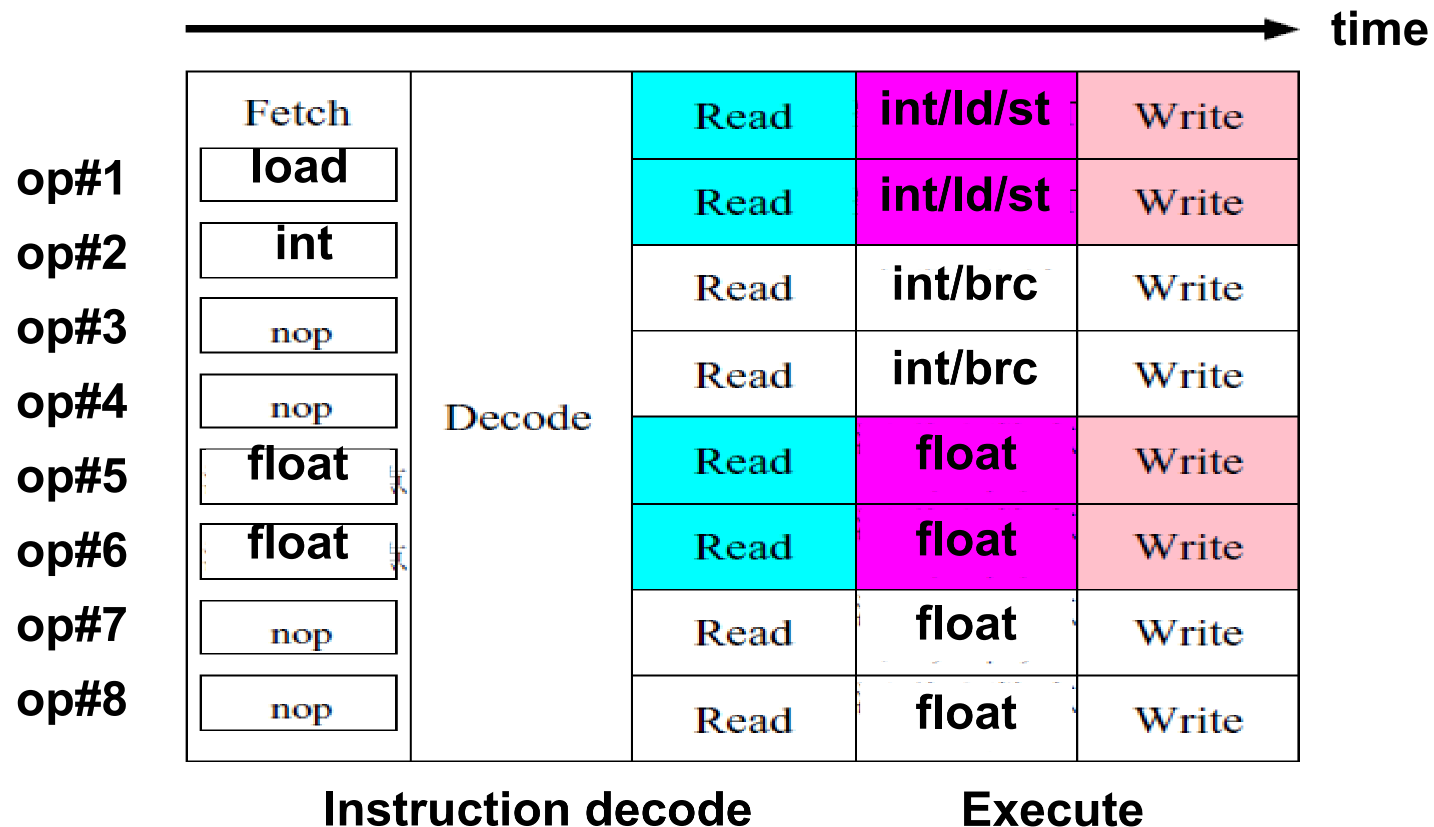
Compiler schedules multiple instructions in parallel

NAIST

VLIW (very long instruction word)

Compiler statically schedules instructions in long-format.

- Binary code is dedicated to specific parallel architecture.
- Simple hardware can reduce power consumption.



NAIST

Example of scheduling

【Livermore14ループ】

```
   DO 1 K=1,400
 1 X(K)=Q+Y(K)*(R*Z(K+10)+T*Z(K+11))
```

4重ループアンローリング
ソフトウェアパイプライニング

```
L:add  Z()+4>Z()
  ld   Z(K+10)
  ld   Z(K+11)
  ld   Z(K+12)
  ld   Z(K+13)
  ld   Z(K+14)
```

```
fmul  R*Z(K+10)>a
fmul  R*Z(K+11)>b
fmul  R*Z(K+12)>c
fmul  R*Z(K+13)>d
fmul  T*Z(K+11)>e
fmul  T*Z(K+12)>f
fmul  T*Z(K+13)>g
fmul  T*Z(K+14)>h
fadd  a+e>i
fadd  b+f>j
fadd  c+g>k
fadd  d+h>l
```

```
  add  X()+4>X()
  cmp  K,400

  brc  1,L
```

```
ld    Y(K  )
ld    Y(K+1)
ld    Y(K+2)
ld    Y(K+3)
add   Y()+4>Y()
fmul  Y(K  )*i>m
fmul  Y(K+1)*j>n
fmul  Y(K+2)*k>o
fmul  Y(K+3)*l>p
fadd  Q+m>X(K  )
fadd  Q+n>X(K+1)
fadd  Q+o>X(K+2)
fadd  Q+p>X(K+3)
st    X(K  )
st    X(K+1)
st    X(K+2)
st    X(K+3)
```


NAIST.

Packing of instructions

2	L:ld Z(K+10)	fmul Y(K)*i>m	
2	ld Z(K+11)	fmul Y(K+1)*j>n	
2	ld Z(K+12)	fmul Y(K+2)*k>o	
2	ld Z(K+13)	fmul Y(K+3)*l>p	
2	ld Z(K+14)	fadd Q+m>X(K)	fmul R*Z(K+10)>a
3	add Z()+4>Z()	fadd Q+n>X(K+1)	fmul R*Z(K+11)>b
3	add Y()+4>Y()	fadd Q+o>X(K+2)	fmul R*Z(K+12)>c
2	ld Y(K+3)	fadd Q+p>X(K+3)	fmul R*Z(K+13)>d
2		st X(K)	fmul T*Z(K+11)>e
2		st X(K+1)	fmul T*Z(K+12)>f
2		st X(K+2)	fmul T*Z(K+13)>g
2		st X(K+3)	fmul T*Z(K+14)>h
4	add X()+4>X()	ld Y(K)	fadd a+e>i
4	cmp K,400	ld Y(K+1)	fadd b+f>j
4		ld Y(K+2)	fadd c+g>k
E	brc 1,L		fadd d+h>l

VLIW: 38operations, 128bytes
Superscalar: 38operations, 152bytes

Instruction scheduling

Wide-issue VLIW requires global scheduling beyond basic blocks.

- **Software pipelining**
- **Loop unrolling**
- **Trace scheduling**
- **Percolation scheduling**

NAIST.

Software pipelining

- Scheduling for eliminating bubbles in pipeline

- Consider latency of each instruction.

```
ld   r1,4,r8          addr (r1+4) ⇒ r8
 1cycle bubble
add  r8,8,r10         add
st   r1,4,r10         r10 ⇒ addr (r1+4)
ld   r1,r5<<2,r11     addr (r1+r5*4) ⇒ r11
 1cycle bubble
sub  r10,r11,r12      subtract
st   r1,r5<<2,r12     r12 ⇒ addr (r1+r5*4)
```

- Rescheduling can reduce execution time.

```
ld   r1,4,r8
ld   r1,r5<<2,r11
add  r8,8,r10
sub  r10,r11,r12
st   r1,4,r10
st   r1,r5<<2,r12
```

Loop unrolling


```
loop: ld    r1,r5<<2,r8       addr (r1+r5*4) ⇒ r8
       1cycle bubble
      fadd r8,r9,r10          floating add
       2cycles bubble
      st    r1,r5<<2,r10      r8 ⇒ addr (r1+r5*4)
      add   r5,1,r5           update index
      comp r5,30              repeat 30 times ?
      bl    loop
```

- **N-unrolling needs many registers but can increase speed. (case N=3)**

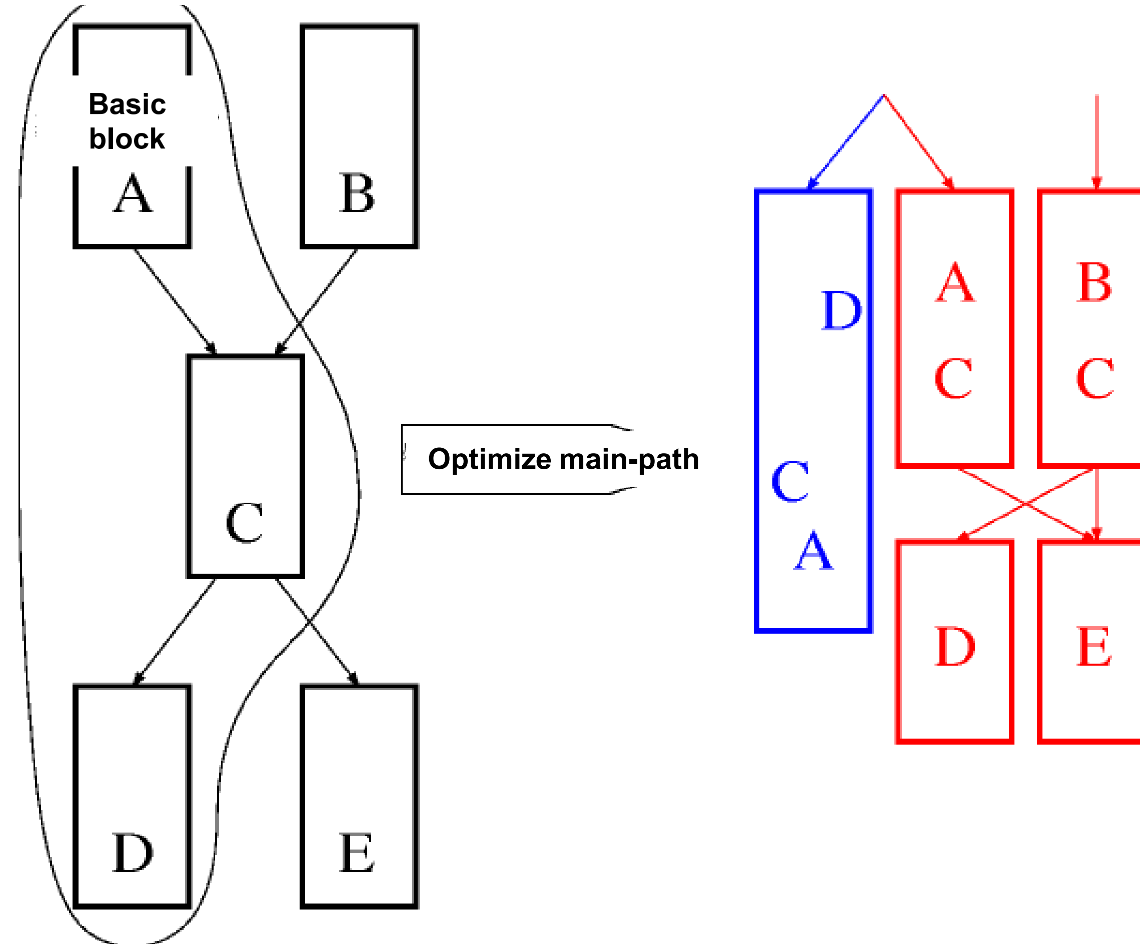
```
loop: ld    r1,r5<<2,r8
      ld    r2,r5<<2,r12      assume r2 = r1+4
      ld    r3,r5<<2,r16      assume r3 = r1+8
      fadd r8,r9,r10
      fadd r12,r9,r13
      fadd r16,r9,r17
      st    r1,r5<<2,r10
      st    r2,r5<<2,r13
      st    r3,r5<<2,r17
      add   r5,3,r5
      comp r5,30
      bl    loop
```


NAIST.

Trace scheduling

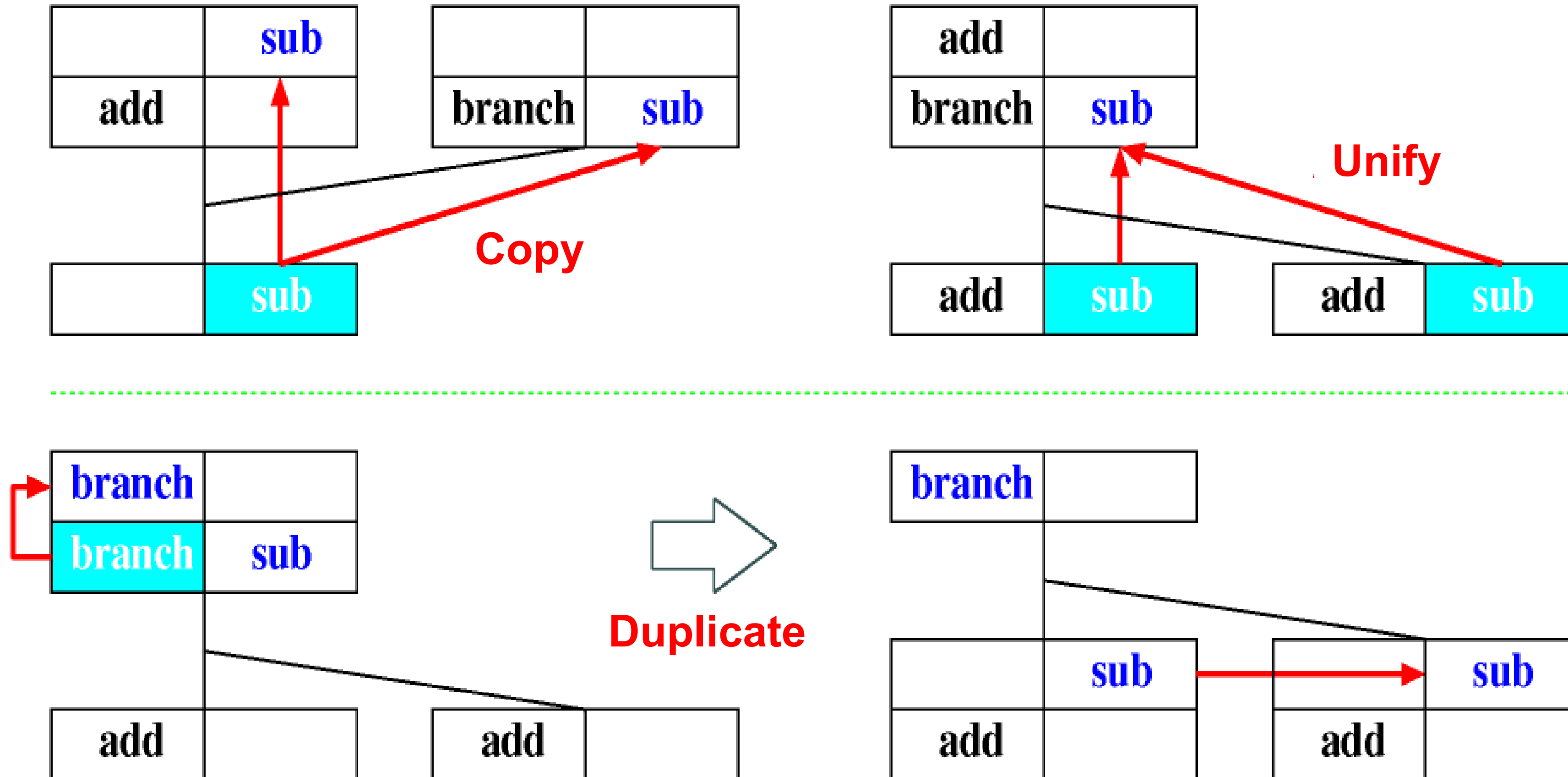
- Get frequently executed path by trial-execution.
- Schedule instructions in main-path beyond basic-block.

  Code size often explodes with many basic-blocks.



Percolation scheduling

Suppress code size explosion to same extent.



NAIST.

Speculation by compiler

- Superscalar has complicated hardware for speculation.
- If compiler does such aggressive scheduling as superscalar, hardware can be simplified.
- However, compiler cannot take risk to crash user programs by illegal memory access.

- Compiler needs special instructions that can suppress illegal memory access.

Speculation by compiler

"Non-faulting load/execution" and "checking store" instructions for VPP5000 supercomputer

Source program	Normal insn	With speculation support
		`dld a -> r1` (speculation)
		`dld b -> r2` (speculation)
		`dadd r1+r2 -> r3` (speculation)
`if (cond) {`	`branch on cond`	`branch on cond`
`*x = *a + *b;`	`ld a`	`cst r3 -> x` (check)
	`ld b`	
	`add a+b`	
`}`	`st x`	

"Non-faulting load/check" instructions for IA64 (intel)

```
       ld.s  a    -> r1  (speculation)
       branch on cond
       chk.s r1,retry
ret:   ...
retry: ld    a    -> r1  (retry)
       branch ret
```


NAIST.

Summary

Compiler is conservative if no architectural support for speculation.

Architects should consider special hardware to support compilers to make hardware more simple.

That's all for today