Computer Systems Design

Lesson 11

Control flow optimization

Alexander Antonov, Assoc. Prof., ITMO University

Hangzhou, 2025

ITMO

Outline the lesson

- Generic strategies of data directed control flow implementation
- Control flow implementations in pure hardware
- Control flow implementations in processor microarchitecture
- Control flow optimizations in compilers
- Control flow optimizations in programming

Control and Data Flow Graph

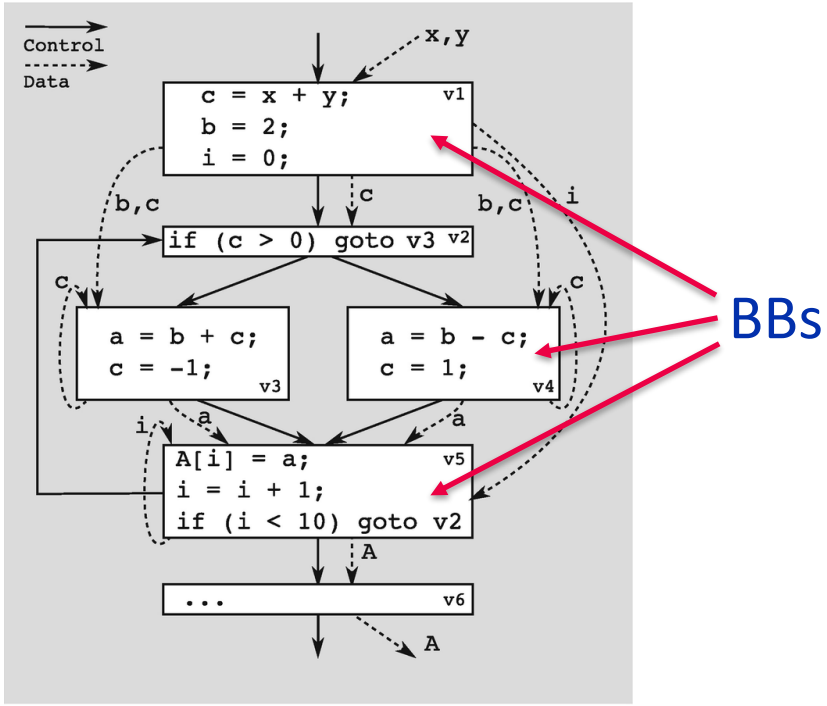
CDFG (Control and Data Flow Graph) – common IR for computer programs in compilers

Basic blocks (BBs) have one entry and one exit point

Optimizations are split in applied:

- ***within BBs*** (data flow optimizations)
- ***between BBs*** (control flow optimizations)

```
1: ...
2: c = x + y;
3: b = 2;
4: for (i = 0; i < 10; i++) {
5:     if (c > 0) {
6:         a = b + c;
7:         c = -1;
8:     } else {
9:         a = b – c;
10:        c = 1;
11:    }
12:    A[i] = a;
13: }
14: ...
```



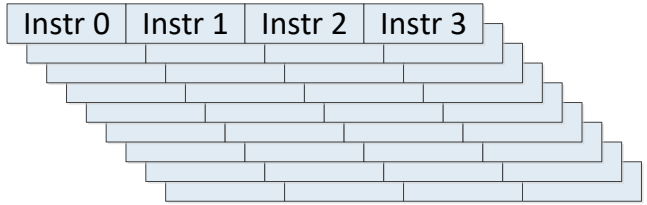
Data-directed control flow

Data-directed control flow: inherent part of many complex algorithms, difficult to optimize

Generic implementation strategies:

- ***wait*** until condition resolved, then start branch execution
- ***if-conversion***: pre-compute all possible outcomes, then select actual ones
  converts control dependencies into data dependencies
- ***restructure*** execution for extraction of parallelism
- ***predict*** most likely condition(s), ***flush*** computations if mispredicted

```
x = a + b;
if (x > 100) {
    out = x - 10;
} else {
    out = x * x;
}
return out;
```

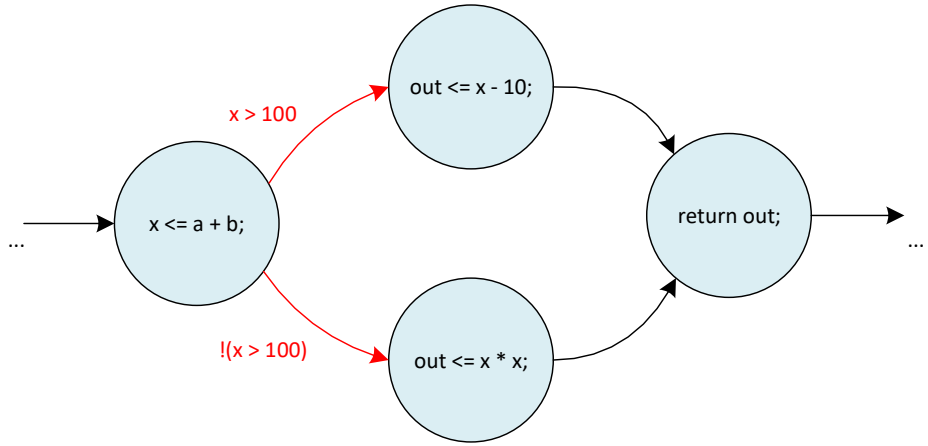


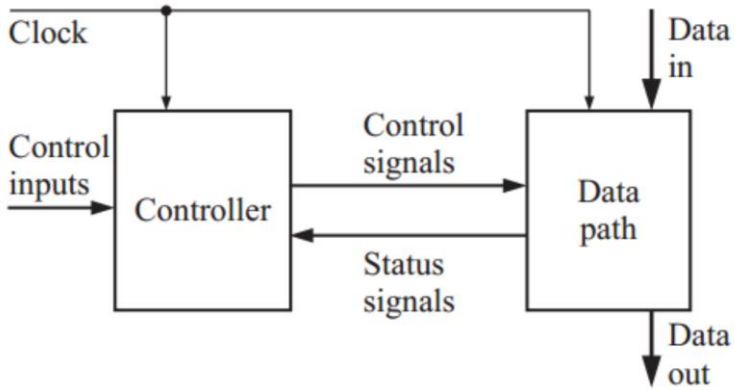
ITMO

Pure hardware

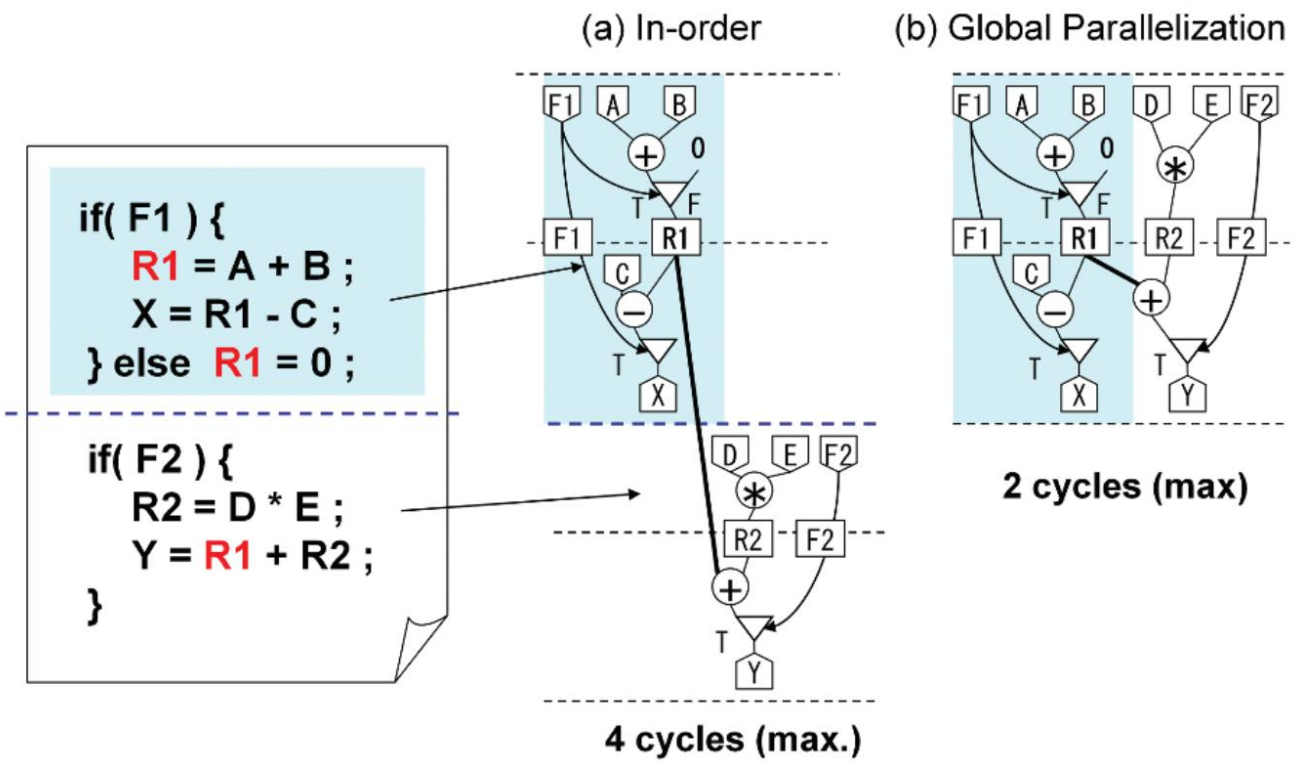
Waiting until condition resolved: finite state machines (FSM)

```
x = a + b;
if (x > 100) {
    out = x - 10;
} else {
    out = x * x;
}
return out;
```





“If-conversion”: pre-computed data multiplexing

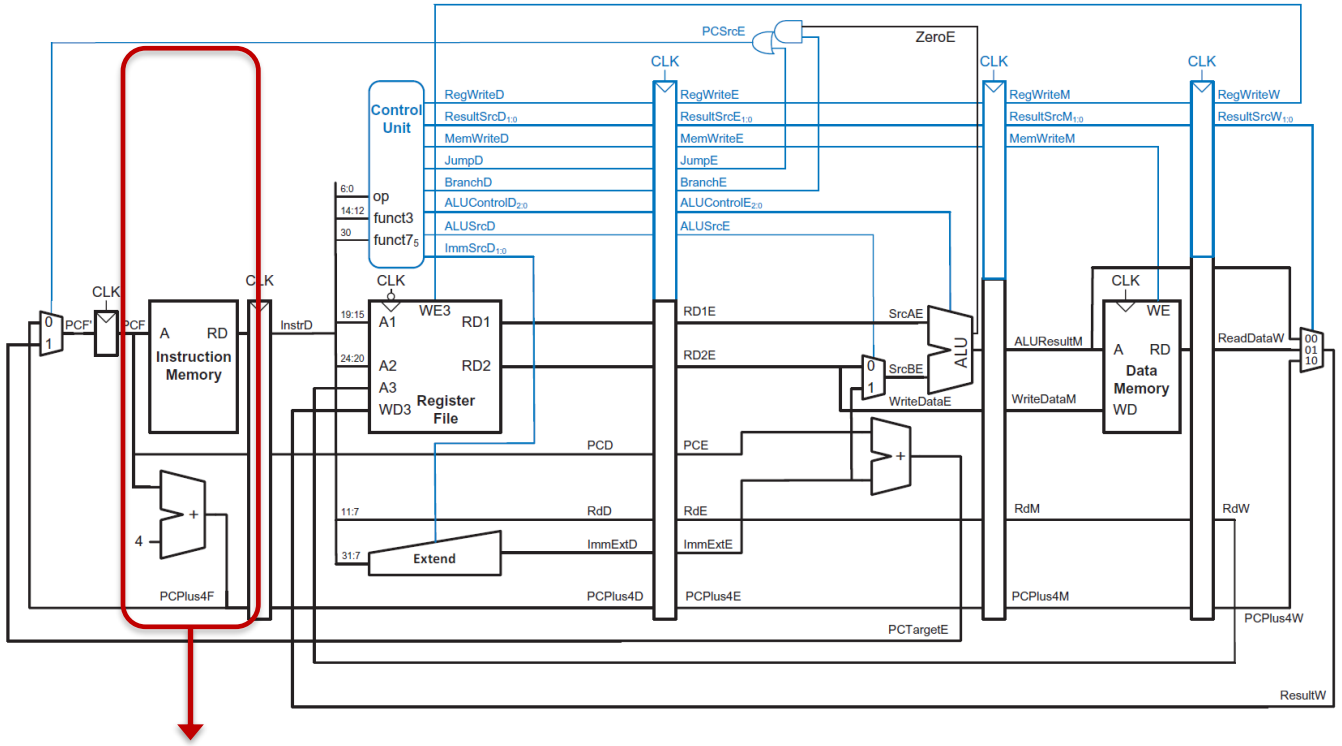


Bad: redundant computations (unused in active branches)

ITMO

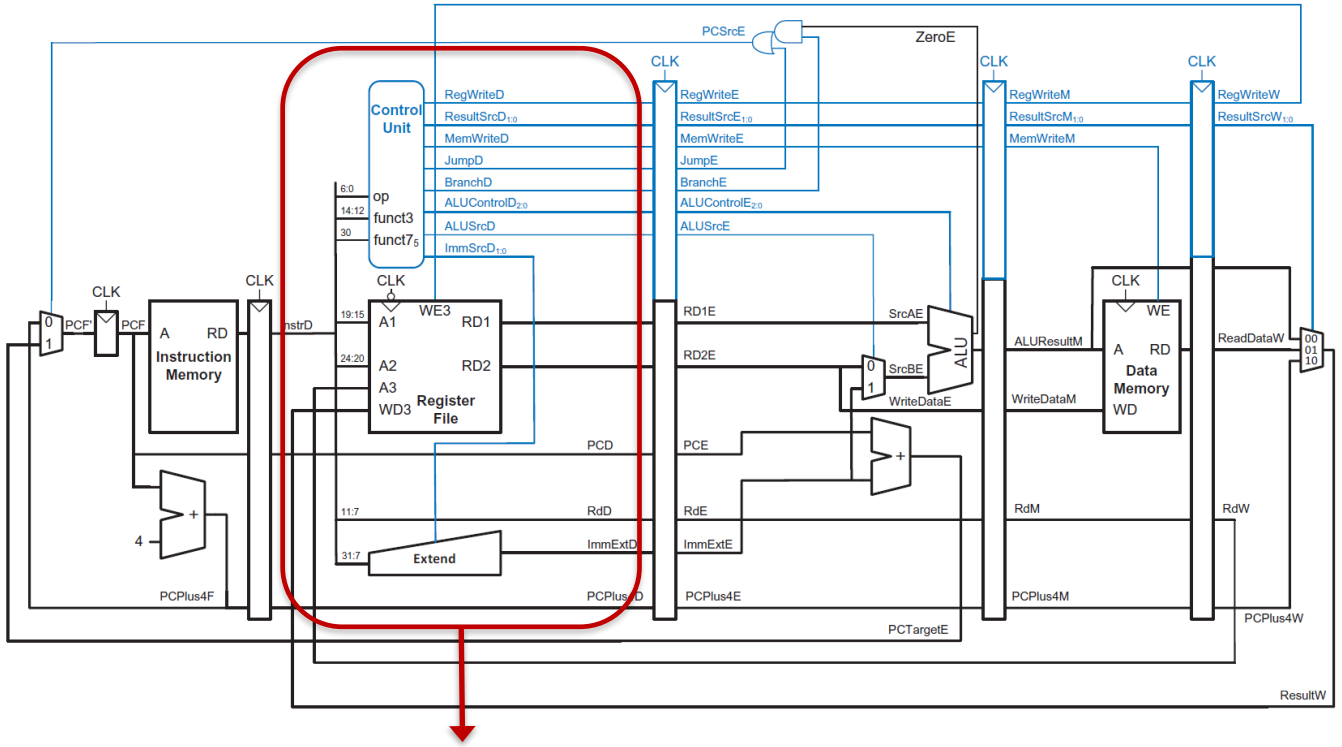
Processor microarchitecture

Branches in classic RISC pipeline (IF stage)



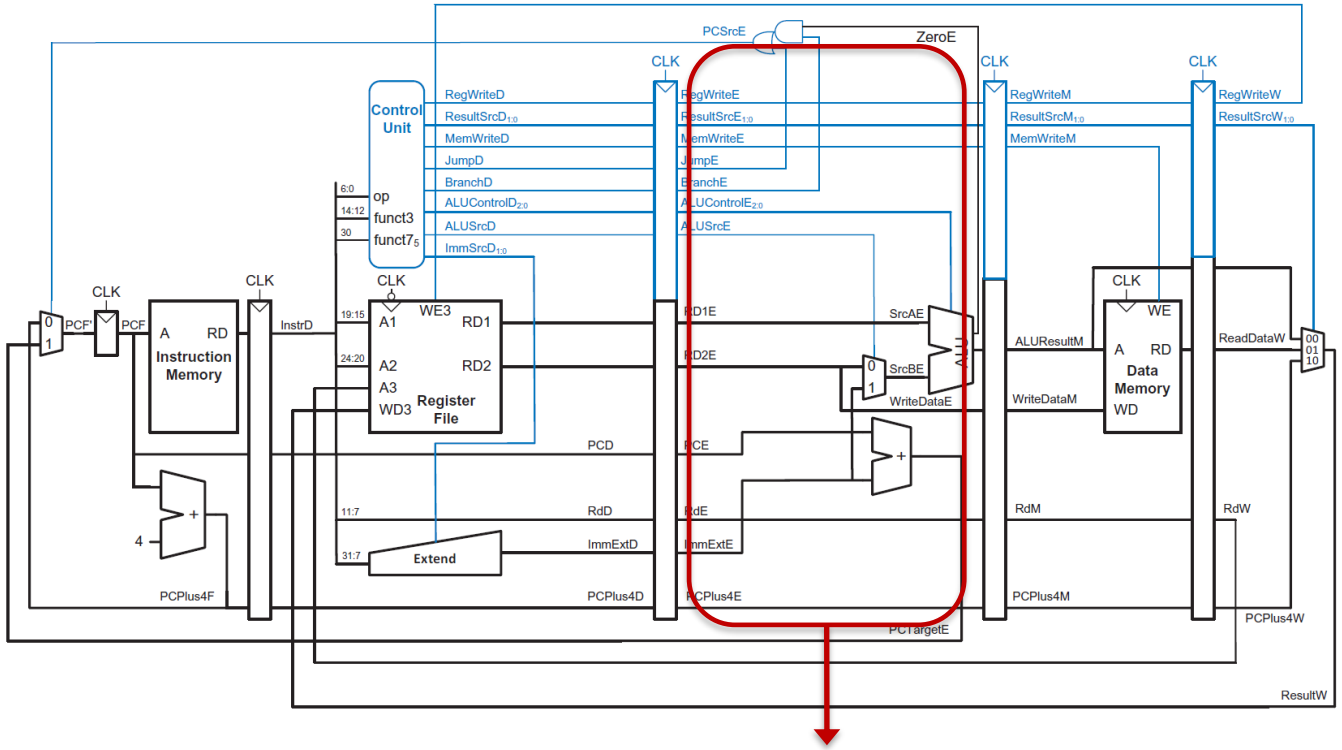
Instruction type (branch?): unknown
Branch condition (for conditional): unknown
Static branch target: unknown
Dynamic (computed) branch target: unknown

Branches in classic RISC pipeline (ID stage)



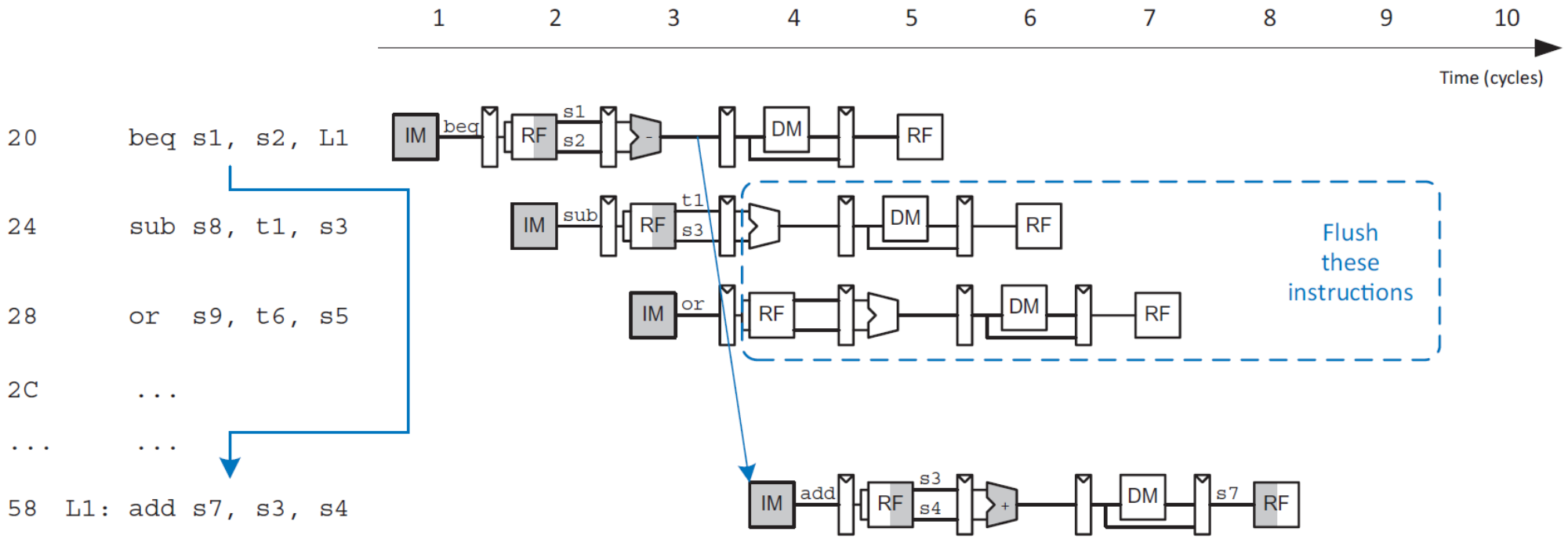
Instruction type (branch?): known
Branch condition (for conditional): unknown
Static branch target: known
Dynamic (computed) branch target: unknown

Branches in classic RISC pipeline (EX stage)



Instruction type (branch?): known
Branch condition (for conditional): known
Static branch target: known
Dynamic (computed) branch target: known

Flushing wasted instructions in classic RISC pipeline



ITMO

Instruction flow speculation: dynamic branch prediction

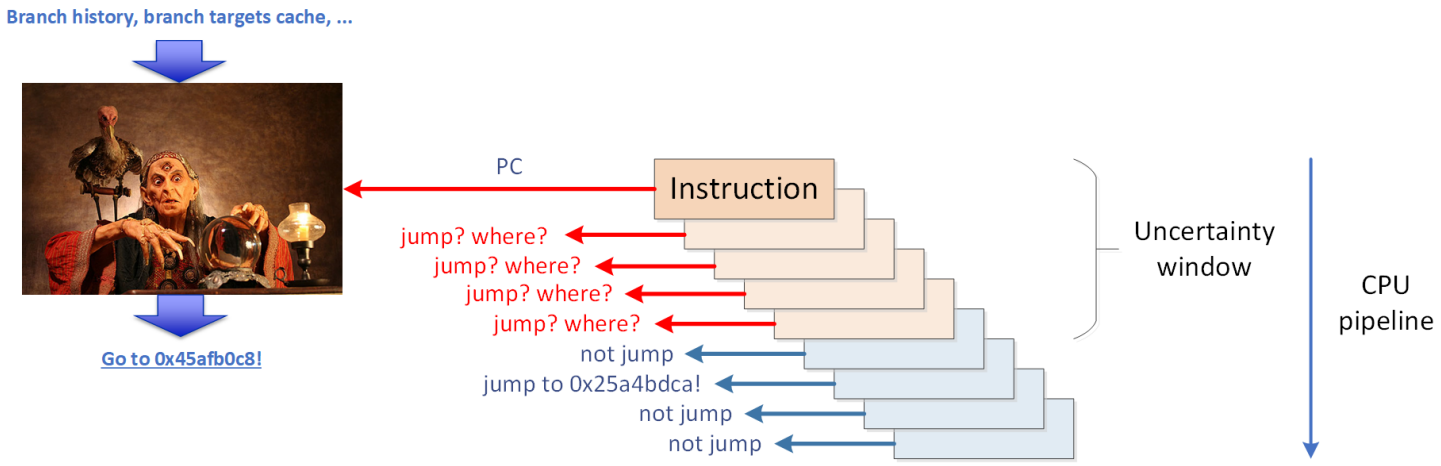
CPU pipelines require ***early decision making about instruction control transfers***

dependency: PC write → instruction memory address of the next instruction

But: within instruction flow, there are ***uncertainties*** on early stages:

- instruction type (branch)?
- dynamic branch conditions?
- computed branch targets?

Solution: branch prediction
(without full awareness of actual program state)

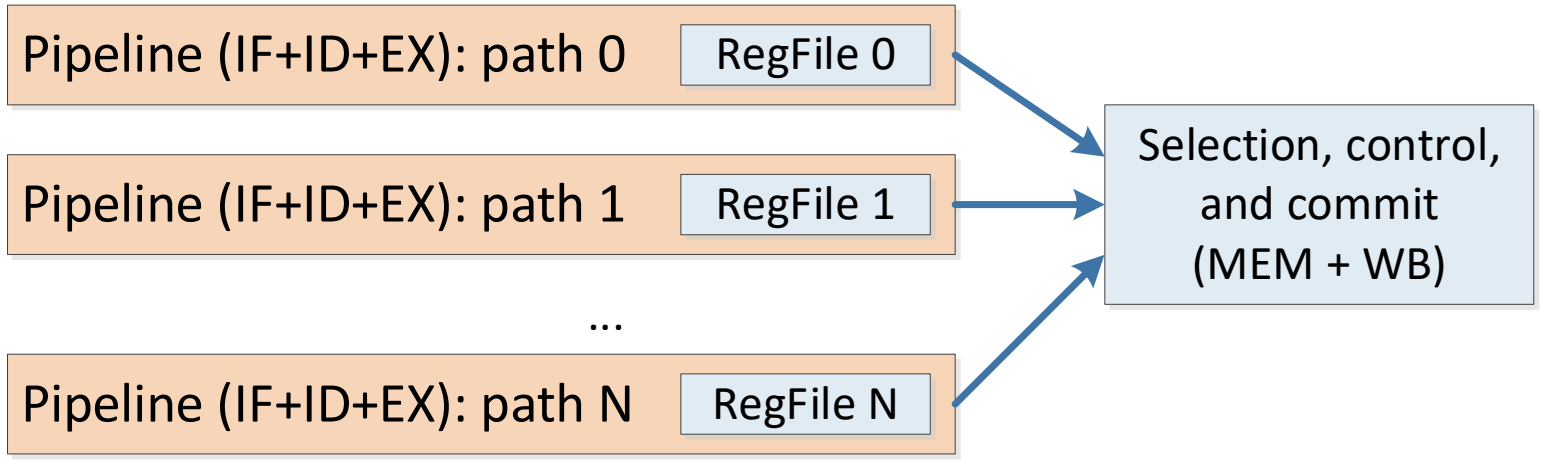


Branches cannot be predicted with 100% accuracy (e.g. during training, due to random data, etc.),
but appear to correlate well with previous branch statistics
More information in Lecture 12

ITMO

Multipath execution

Principle: speculatively fetch and execute ***several paths simultaneously***, continuously discard false branches



Bad: redundant computations and storage

Issue: nested branching in speculated paths

ITMO

Exceptions and interrupts in processors

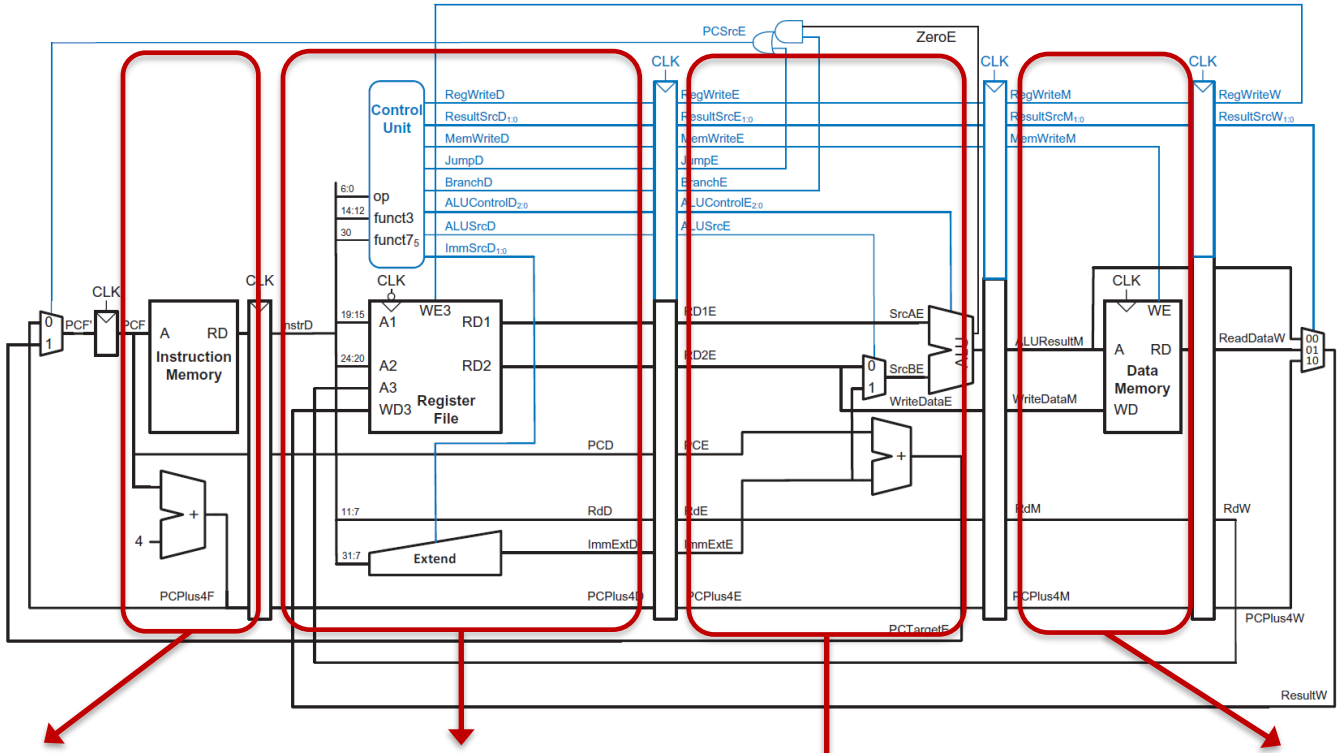
In addition to branches, the following events can alter control flow:

- ***Exception:*** event caused by abnormal program execution
- ***Interrupt:*** event fired by external device

When occurred, control flow is transferred to predefined location where ***software handler*** is expected to reside.

Exceptions and interrupts in classic RISC pipeline

Earlier exceptions have ***higher*** priority for each instruction



Out-of-range address
Page fault
No rights for execution
Incorrect alignment

Incorrect instruction code

Arithmetic overflow/underflow

Out-of-range address
Page fault
No R/W rights
Incorrect alignment
External interrupts

ITMO

Processor ISA

Predicated instructions

Instructions are annotated with ***predicates*** (conditions), instruction with failed predicate = no

Concept similar to “if-conversion”

Source code

```
if (emp_status == ACTIVE) {
    active_emps++;
    total_payroll += emp_pay;
}
else {
    inactive_emps++;
}
```

Conventional generated code (with branches)

```
{
    cmp.ne p1 = rs, ACTIVE    // compare emp_status
    (p1) br else              // jump to else code if condition fails
}
.label then
{
    add  rt = rt, rp          // sum total_payroll + emp_pay
    add  ra = ra, 1           // increment active_emps
    br  join
}
.label else
{
    add  ri = ri, 1           // increment inactive_emps
}
.label join
```

Code with predicated execution

```
{
    cmp.eq  p1, p2 = rs, ACTIVE  // compare emp_status
} {
    (p1)  add rt = rt, rp        // sum total_payroll + emp_pay
    (p1)  add ra = ra, 1         // increment active_emps
    (p2)  add ri = ri, 1         // increment inactive_emps
}
```

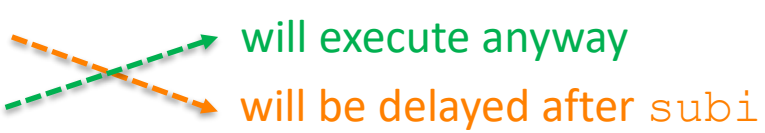
Restructured branches in ISA: delay slots

ISA feature making ***N instructions after branch execute anyway***

Reduces mispredict penalty

Considered obsolete (difficult to use, difficult to implement in OoO processors)

```
lw r2, r1, 0x100
add r4, r2, r0
beq r1, r10, 0x14
subi r6, r5, 0x20
mul r10, r10, r11
```



GCC:

`-fdelayed-branch`

If supported for the target machine, attempt to reorder instructions to exploit instruction slots available after delayed branch instructions.

Enabled at levels `-O1`, `-O2`, `-O3`, `-Os`, but not at `-Og`.

Prediction hints

Some ISAs allocate ***bits*** in instructions to ***hint*** the processor with (un)likely branch occurrence

Reduce misprediction ratio (or can be ignored by processor)

Not widely used in practice

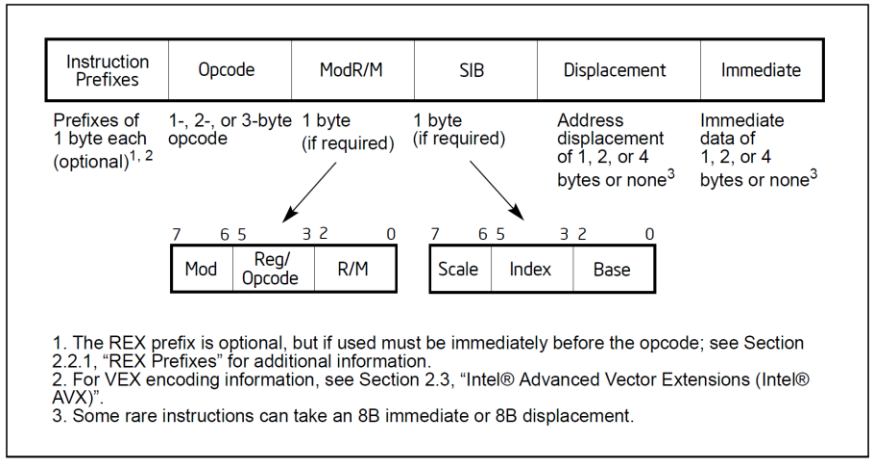


Figure 2-1. Intel 64 and IA-32 Architectures Instruction Format

INSTRUCTION FORMAT

- BNDCFGU.EN and/or IA32_BNDCFGS.EN is set.
- When the F2 prefix precedes a near CALL, a near RET, a near JMP, or a near Jcc instruction (see Chapter 17, "Intel® MPX," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*).

- Group 2
  - Segment override prefixes:
    - 2EH—CS segment override (use with any branch instruction is reserved).
    - 36H—SS segment override prefix (use with any branch instruction is reserved).
    - 3EH—DS segment override prefix (use with any branch instruction is reserved).
    - 26H—ES segment override prefix (use with any branch instruction is reserved).
    - 64H—FS segment override prefix (use with any branch instruction is reserved).
    - 65H—GS segment override prefix (use with any branch instruction is reserved).
  - Branch hints[1]:
    - 2EH—Branch not taken (used only with Jcc instructions).
    - 3EH—Branch taken (used only with Jcc instructions).
- Group 3
  - Operand-size override prefix is encoded using 66H (66H is also used as a mandatory prefix for some instructions).
- Group 4
  - 67H—Address-size override prefix.

Intel® 64 and IA-32 Architectures Developer's Manual: Vol. 2A

ITMO

Compilers and programming

ITMO

If-conversion in SW: branchless programming

Has branches

Unfriendly to pipelining

```
if (a > b) {
    return a;
} else {
    return b;
}
```

No branches

Friendly to pipelining

```
return (a > b) * a + (a <= b) * b;
```

Advanced branchless programming: function tables

```c
#include <stdio.h>
#include <stdlib.h>

void fun0(int a)
{
    printf("Fun0: a: %d\n", a);
}

void fun1(int a)
{
    printf("Fun1: a: %d\n", a);
}

void fun2(int a)
{
    printf("Fun2: a: %d\n", a);
}

void call_fun(int fun_num, int a) {     // function caller
    switch (fun_num) {
        case 0: fun0(a); break;
        case 1: fun1(a); break;
        case 2: fun2(a); break;
    }
}


int main()
{
    // calling functions
    call_fun(0, 10);
    call_fun(1, 11);
    call_fun(2, 12);

    return 0;
}
```

Many branches
(in call_fun function)

```c
#include <stdio.h>
#include <stdlib.h>

void fun0(int a)
{
    printf("Fun0: a: %d\n", a);
}

void fun1(int a)
{
    printf("Fun1: a: %d\n", a);
}

void fun2(int a)
{
    printf("Fun2: a: %d\n", a);
}

void (*fun_ptr[3])(int);        // table of functions

void call_fun(int fun_num, int a) {     // function caller
    (*fun_ptr[fun_num])(a);
}


int main()
{
    // initializing a table of functions
    fun_ptr[0] = &fun0;
    fun_ptr[1] = &fun1;
    fun_ptr[2] = &fun2;

    // calling functions
    call_fun(0, 10);
    call_fun(1, 11);
    call_fun(2, 12);

    return 0;
}
```

Less branches

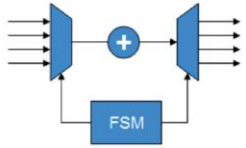
Result:

```
Fun0: a: 10
Fun1: a: 11
Fun2: a: 12
```

Restructuring code for parallel architectures: loop optimizations

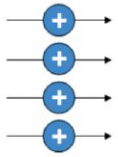
Loop unrolling

```
for (i = 0; i<4; i++)
{
   r[i] = a[i] + b[i];
}
```



No Unrolling
1 Adder shared for 4 additions
Latency = 4 cycles

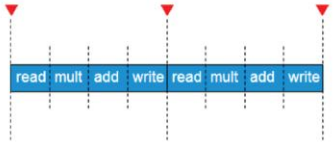
```
r[0] = a[0] + b[0];
r[1] = a[1] + b[1];
r[2] = a[2] + b[2];
r[3] = a[3] + b[3];
```



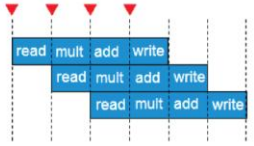
Unrolling = 4 (Full)
4 Adders in parallel
Latency = 1 cycle

Loop pipelining

```
for (i = 0; i<3; i++)
{
   out[i] = (in[i] * coef1) + coef2;
}
```



No Pipelining
Latency = 12 cycles
Throughput = 4 cycles



Pipelining Initiation Interval = 1
Latency = 6 cycles
Throughput = 1 cycle

Loop merging

```
for (i = 0; i<32; i++)
{
   a[i] = b[i] * c[i];
}
for (i = 0; i<16; i++)
{
   z[i] = a[i] + x[i];
}
```

No Merging
Loops execute sequentially
Latency = 48 cycles

```
for (i = 0; i<32; i++)
{
   atmp = b[i] * c[i];
   if (i<16)
     z[i] = atmp + x[i];
}
```

Merging Enabled
Loops execute in parallel
Latency = 32 cycles

ITMO

Software prediction hints (GCC)

```
#define likely(x) __builtin_expect(!!(x), 1)
#define unlikely(x) __builtin_expect(!!(x), 0)

if (likely(x > 0)) {
    // This block is expected to execute more frequently
} else {
    // This block is less likely to be executed
}

if (unlikely(error_condition)) {
    // Handle error - expected to be executed infrequently
}
```

Generated machine code will make more likely branches more likely to predict as taken by CPU

ITMO

Summary of control flow implementations

	Hardware	Processor microarchitecture	Processor ISA	Software
Waiting until condition resolved	Finite state machines	Non-pipelined implementations	Most conventional ISAs	Most conventional software
If-conversion	Data pre-computation and multiplexing	Multipath execution	Predicated instructions	If-conversion in compilers, branchless programming
Restructuring execution	Custom hardware parallelization optimizations	Out-of-order execution	Delay slots	Compiler (e.g. loop) optimizations
Prediction	Prediction in hardware logic	Branch prediction	ISA prediction hints	Software prediction hints

Thank you for the lesson!

Alexander Antonov, Assoc. Prof., antonov@itmo.ru

Hangzhou, 2025