Comet Project

A RISC-V processor written in C++ for HLS

Leng-Kai Lin

Outline

- Introduction to Comet
- Implementation
  - Pipeline stages and Pipeline registers
  - Signals
  - Stall, Flush, and Data Forwarding
  - Pipeline stage implementation
  - Performance Counters
- Workflow Demonstration

Outline

- Introduction to Comet
- Implementation
    - Pipeline stages and Pipeline registers
    - Signals
    - Stall, Flush, and Data Forwarding
    - Pipeline stage implementation
    - Performance Counters
- Workflow Demonstration

Introduction to Comet

- A project of INRIA published in IEEE/ACM
  - What You Simulate Is What You Synthesize: Designing a Processor Core from C++ Specifications
- 5-stage pipelined RISC-V processor
  - Classic RISC pipeline: IF / ID / EX / MEM / WB
- Easy to add new functionality
  - Integration of branch predictor (lab)

Outline

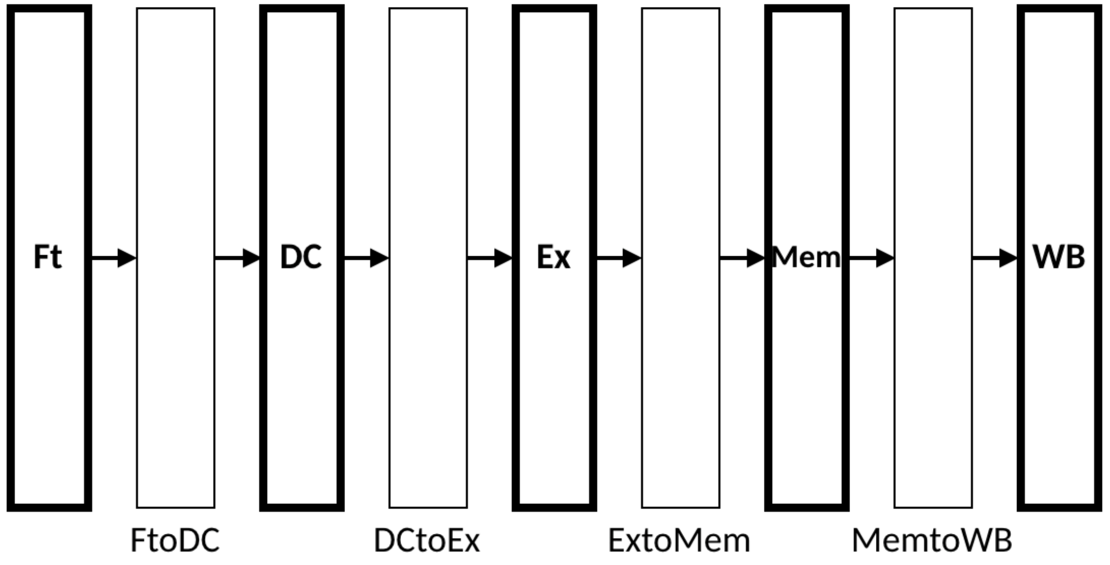
- Introduction to Comet
- Implementation
    - Pipeline stages and Pipeline registers
    - Signals
    - Stall, Flush, and Data Forwarding
    - Pipeline stage implementation
    - Performance Counters
- Workflow Demonstration

Outline

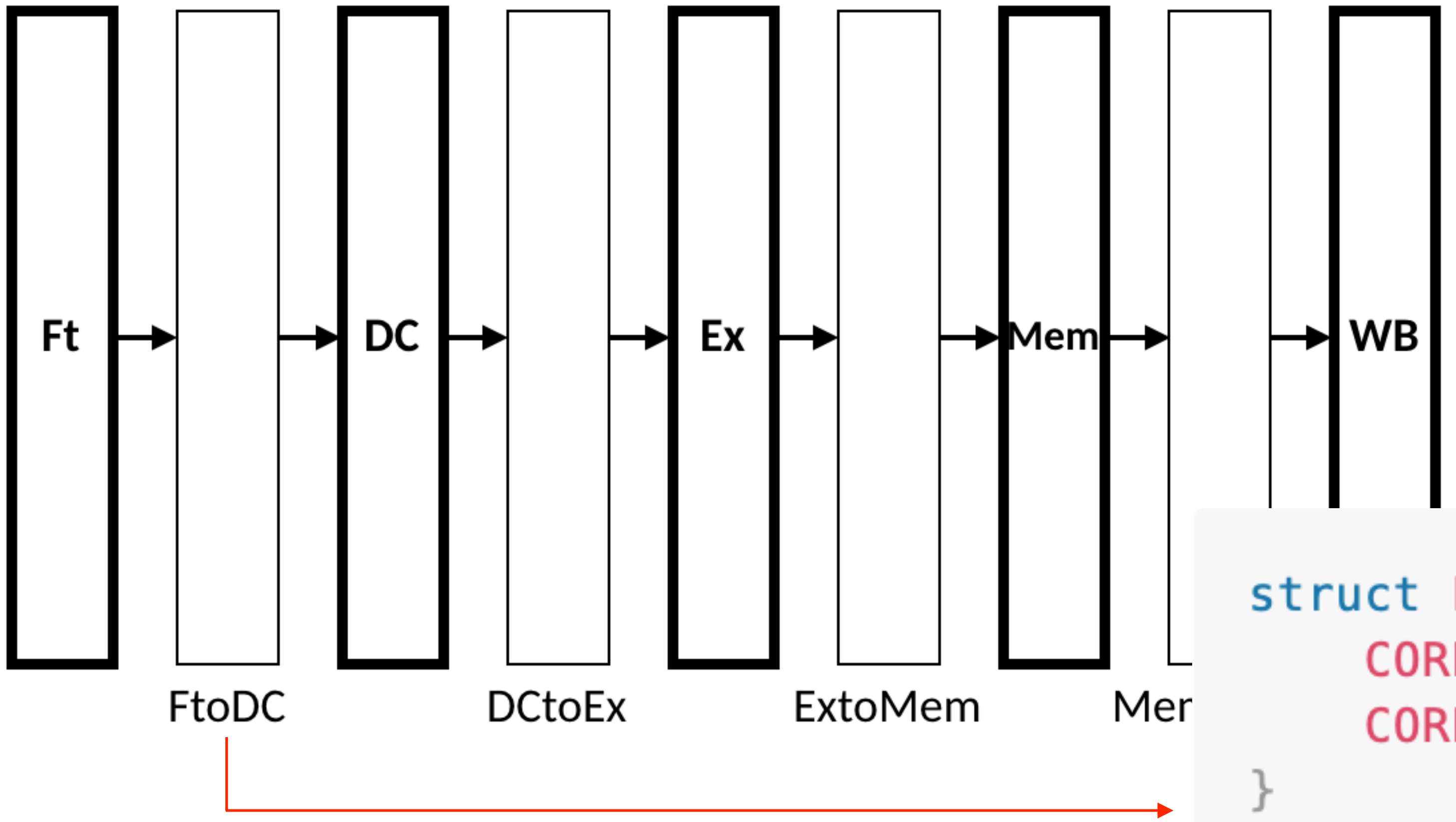
- Introduction to Comet
- Implementation
    - Pipeline stages and Pipeline registers
    - Signals
    - Stall, Flush, and Data Forwarding
    - Pipeline stage implementation
    - Performance Counters
- Workflow Demonstration

Pipeline Registers

- Placed between each pair of pipeline stages
- Declared as C struct, defined in `synthesizable/include/register.h`



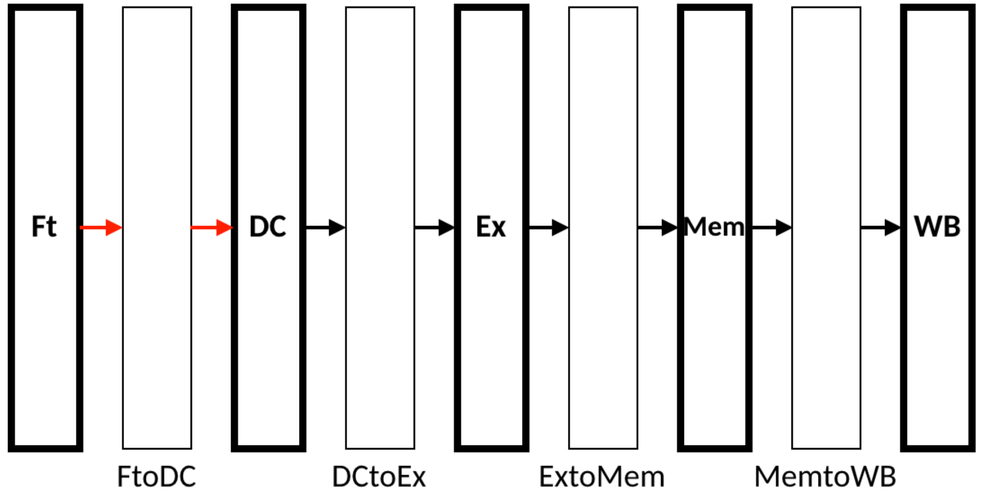
Pipeline Registers



```
struct FtoDC {
    CORE_UINT(32) pc;
    CORE_UINT(32) instruction;
}
```

Pipeline Stages

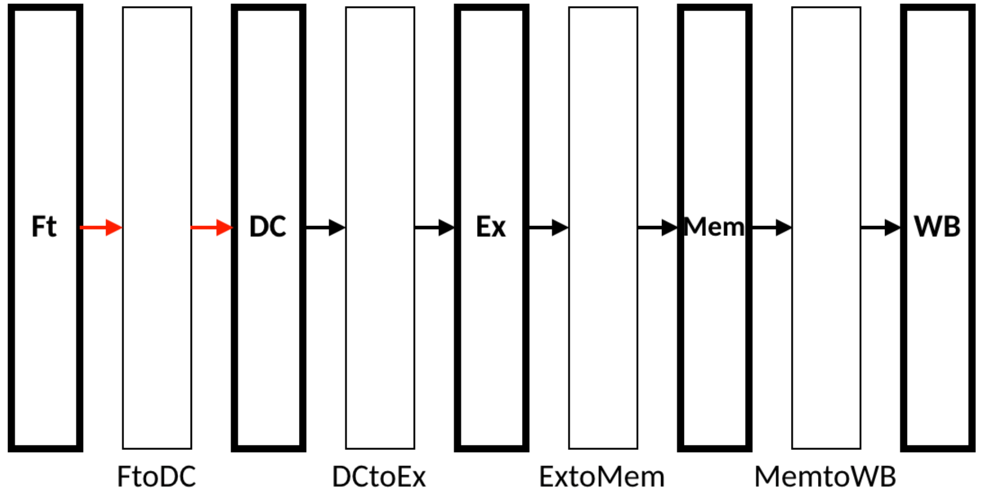
```
struct FtoDC ftoDC;

while (true) {
    Ft(&ftoDC); // Instruction Fetch
    DC(ftoDC);  // Instruction Decode
}
```



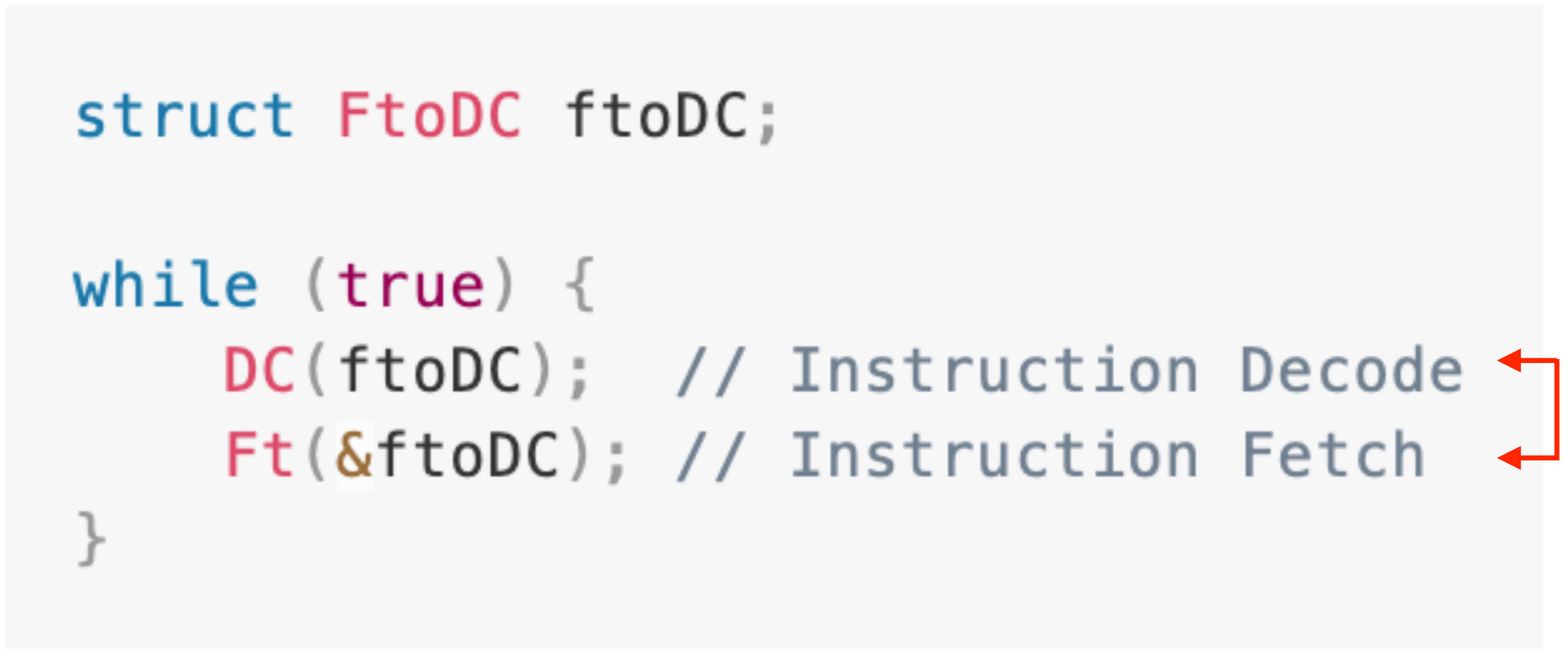
Pipeline Stages

```
struct FtoDC ftoDC;

while (true) {
    Ft(&ftoDC); // Instruction Fetch
    DC(ftoDC);  // Instruction Decode
}
```

dependency

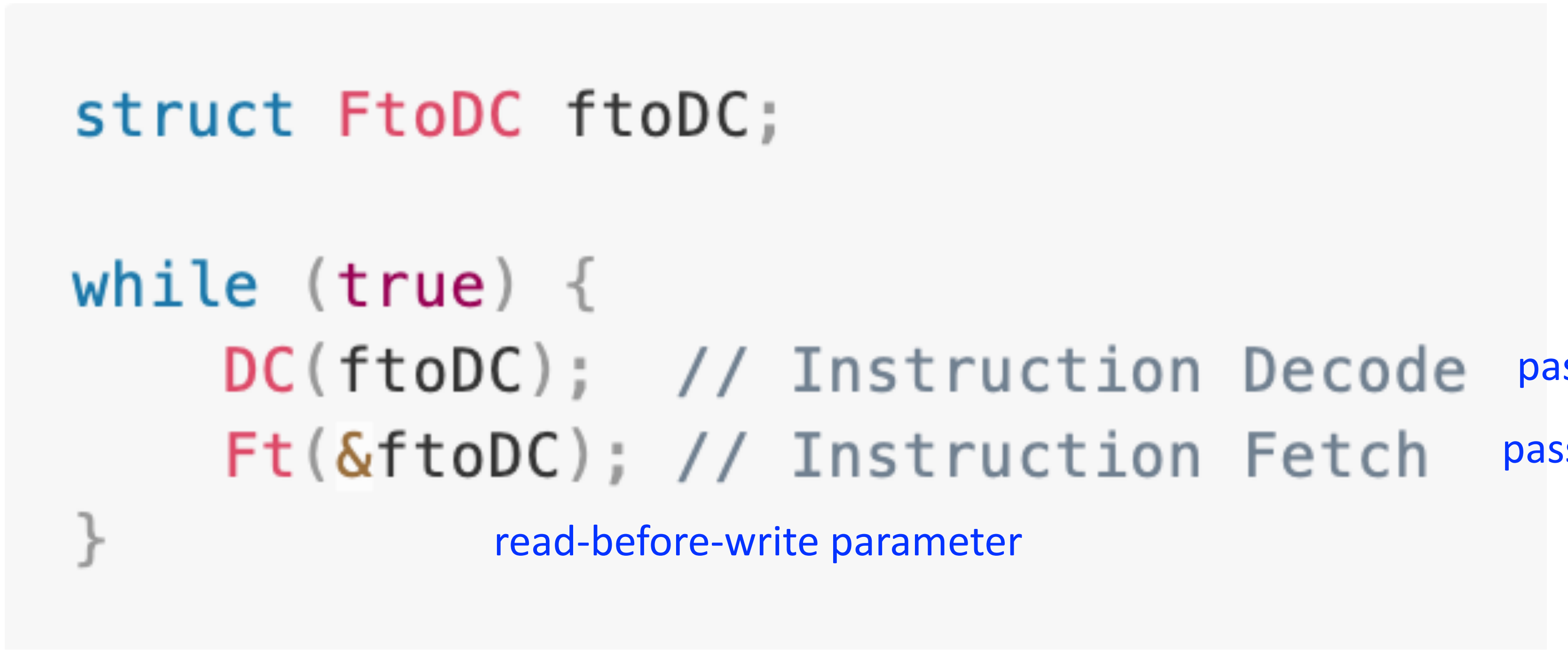


Pipeline Stages



```
struct FtoDC ftoDC;

while (true) {
    DC(ftoDC);  // Instruction Decode
    Ft(&ftoDC); // Instruction Fetch
}
```

Pipeline Stages



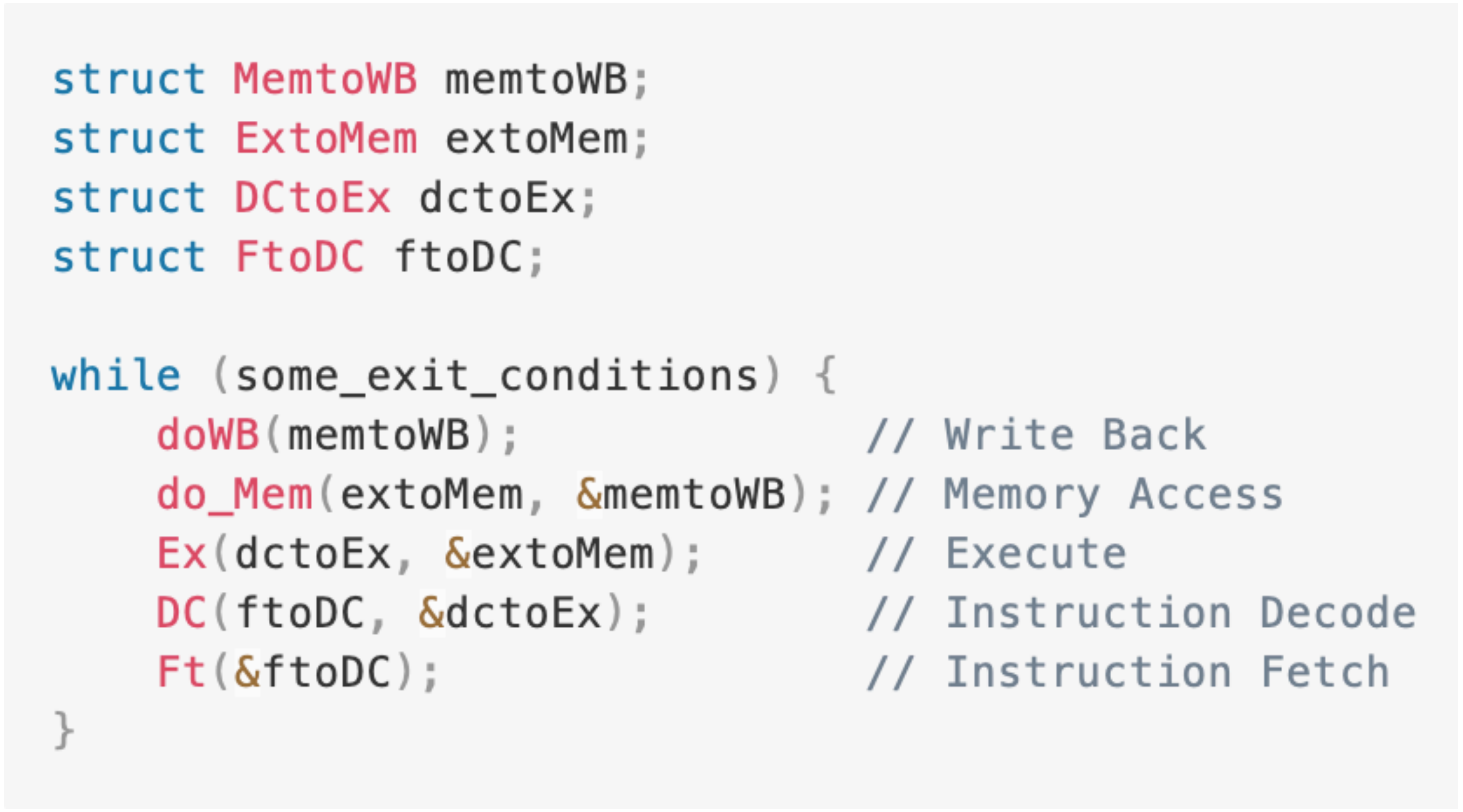
```
struct FtoDC ftoDC;

while (true) {
    DC(ftoDC);  // Instruction Decode
    Ft(&ftoDC); // Instruction Fetch
}
```

passed by value

passed by pointer

read-before-write parameter

Pipeline Stages



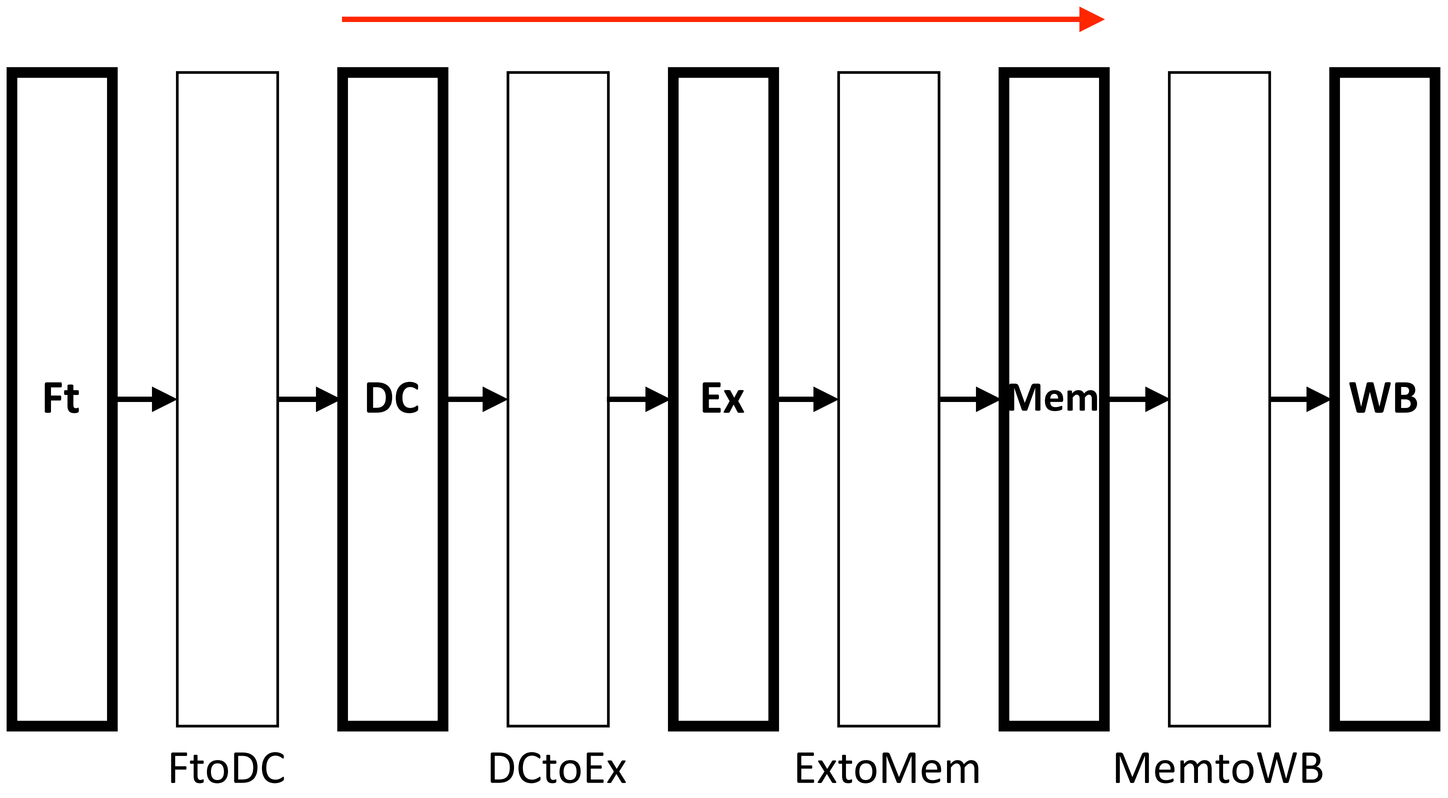
```
struct MemtoWB memtoWB;
struct ExtoMem extoMem;
struct DCtoEx dctoEx;
struct FtoDC ftoDC;

while (some_exit_conditions) {
    doWB(memtoWB);                 // Write Back
    do_Mem(extoMem, &memtoWB);     // Memory Access
    Ex(dctoEx, &extoMem);          // Execute
    DC(ftoDC, &dctoEx);            // Instruction Decode
    Ft(&ftoDC);                    // Instruction Fetch
}
```

Outline

- Introduction to Comet
- Implementation
    - Pipeline stages and Pipeline registers
    - Signals
    - Stall, Flush, and Data Forwarding
    - Pipeline stage implementation
    - Performance Counters
- Workflow Demonstration

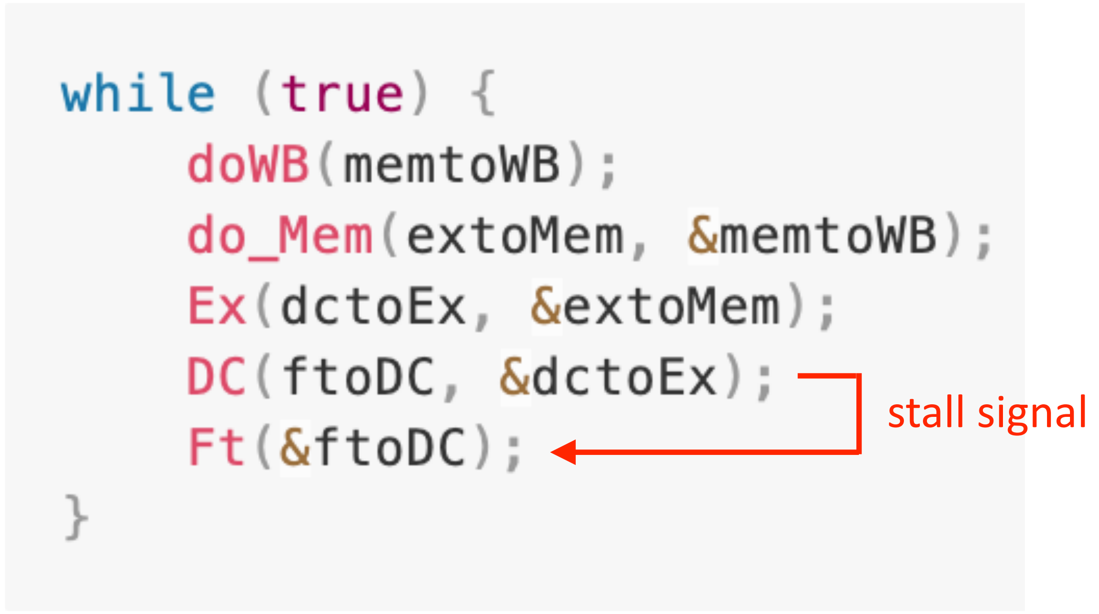
Signals

- Pipeline Registers handle the forward data transmission

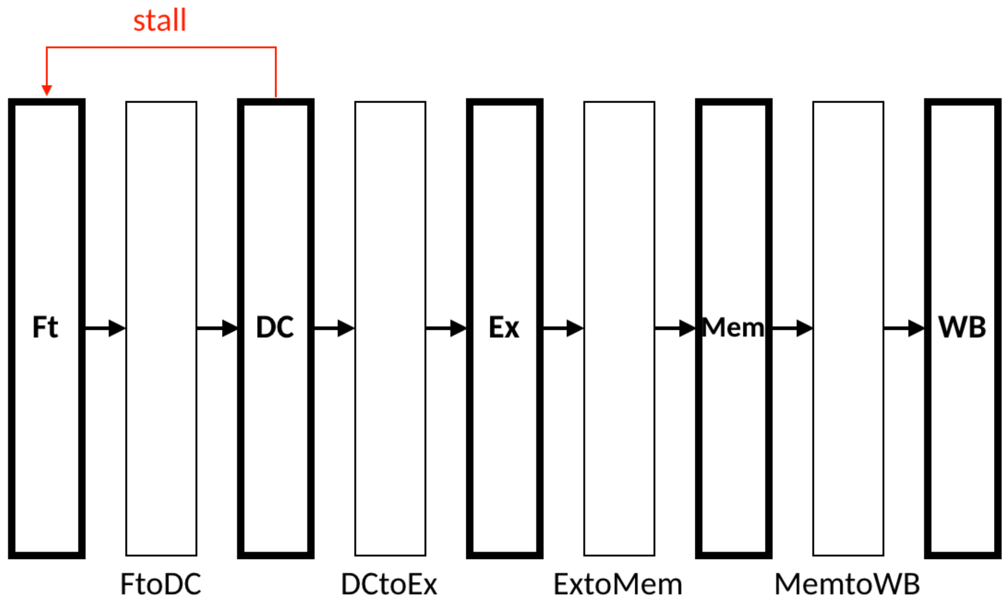


Signals

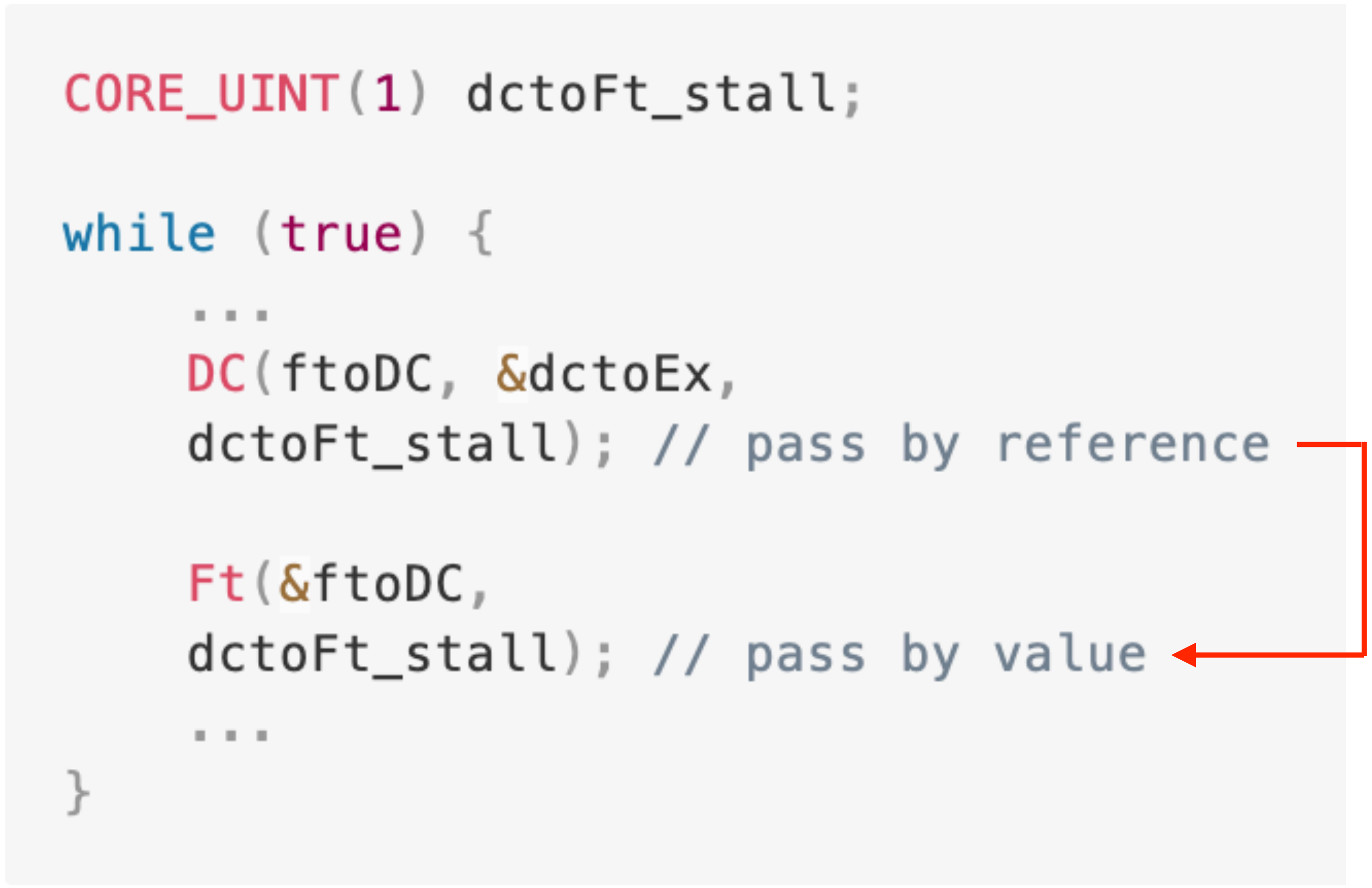
- Backward data transmission also needs to be handled



```
while (true) {
    doWB(memtoWB);
    do_Mem(extoMem, &memtoWB);
    Ex(dctoEx, &extoMem);
    DC(ftoDC, &dctoEx);
    Ft(&ftoDC);
}
```



Signals



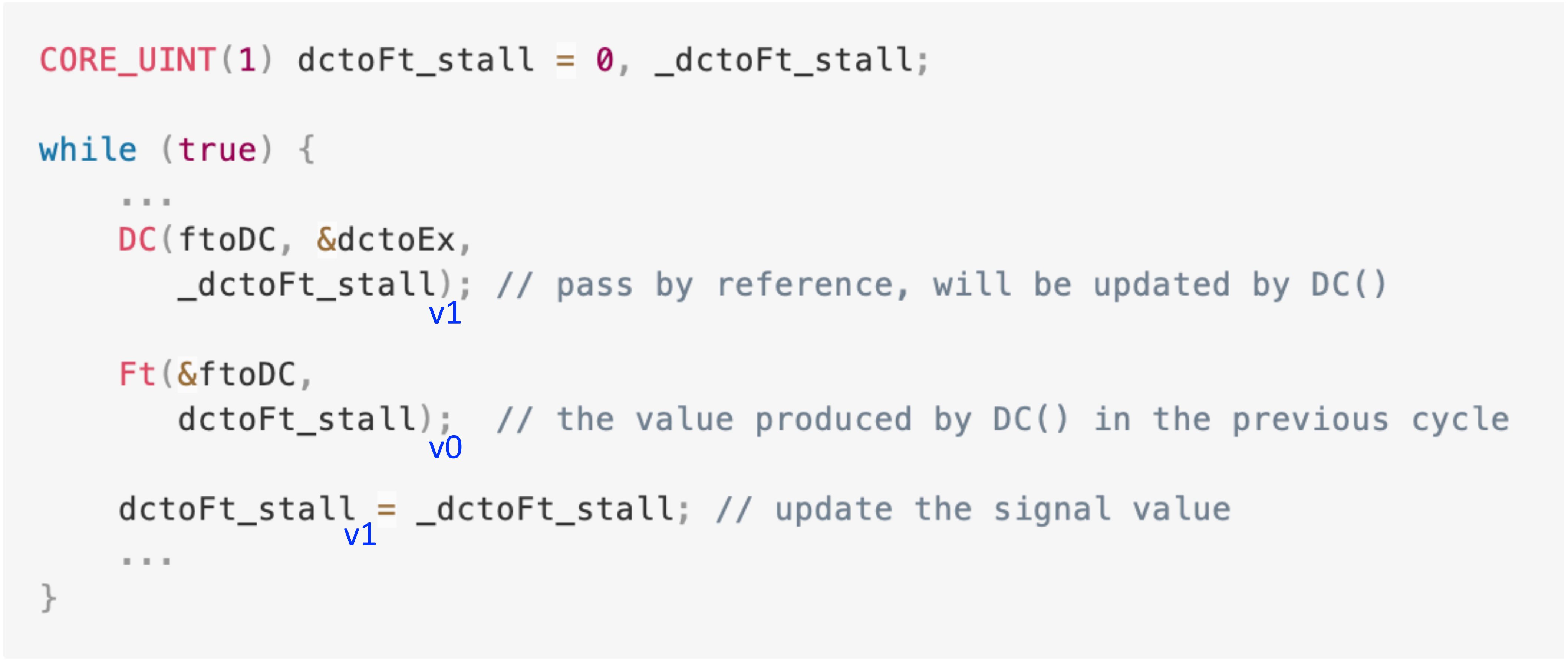
```
CORE_UINT(1) dctoFt_stall;

while (true) {
    ...
    DC(ftoDC, &dctoEx,
    dctoFt_stall); // pass by reference

    Ft(&ftoDC,
    dctoFt_stall); // pass by value
    ...
}
```

dependency

Signals

```
CORE_UINT(1) dctoFt_stall = 0, _dctoFt_stall;

while (true) {
    ...
    DC(ftoDC, &dctoEx,
       _dctoFt_stall); // pass by reference, will be updated by DC()

    Ft(&ftoDC,
       dctoFt_stall);  // the value produced by DC() in the previous cycle

    dctoFt_stall = _dctoFt_stall; // update the signal value
    ...
}
```

Signals



```
CORE_UINT(1) dctoFt_stall = 0, _dctoFt_stall;

while (true) {
    ...
    DC(ftoDC, &dctoEx,
       _dctoFt_stall); // pass by reference, will be updated by DC()
                    v1
    Ft(&ftoDC,
       dctoFt_stall);  // the value produced by DC() in the previous cycle
                    v0
    dctoFt_stall = _dctoFt_stall; // update the signal value
               v1
    ...
}
```

Signals

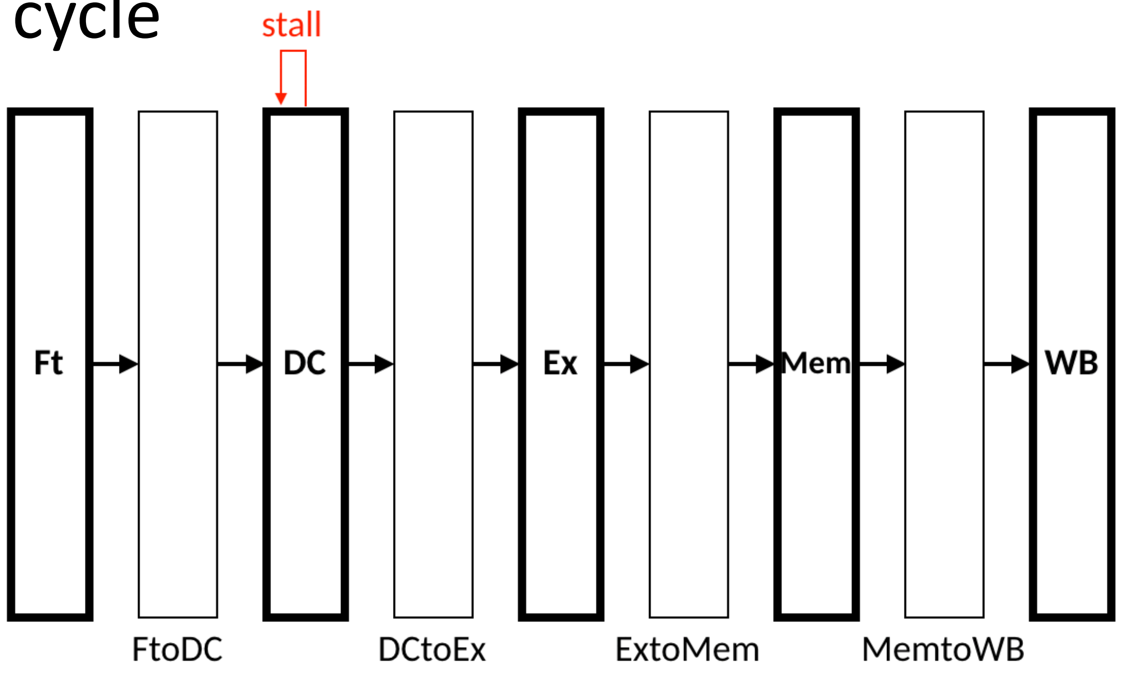
```
CORE_UINT(1) dctoFt_stall = 0, _dctoFt_stall;

while (true) {
    ...
    DC(ftoDC, &dctoEx,
       _dctoFt_stall); // pass by reference, will be updated by DC()
                   v2
    Ft(&ftoDC,
       dctoFt_stall);  // the value produced by DC() in the previous cycle
                   v1
    dctoFt_stall = _dctoFt_stall; // update the signal value
               v2
    ...
}
```

Self Signals

- The DC stage should also inform itself to replace the instruction with the previous instruction in the next cycle

```
while (true) {
    doWB(memtoWB);
    do_Mem(extoMem, &memtoWB);
    Ex(dctoEx, &extoMem);
    DC(ftoDC, &dctoEx);
    Ft(&ftoDC);
}
```

stall signal



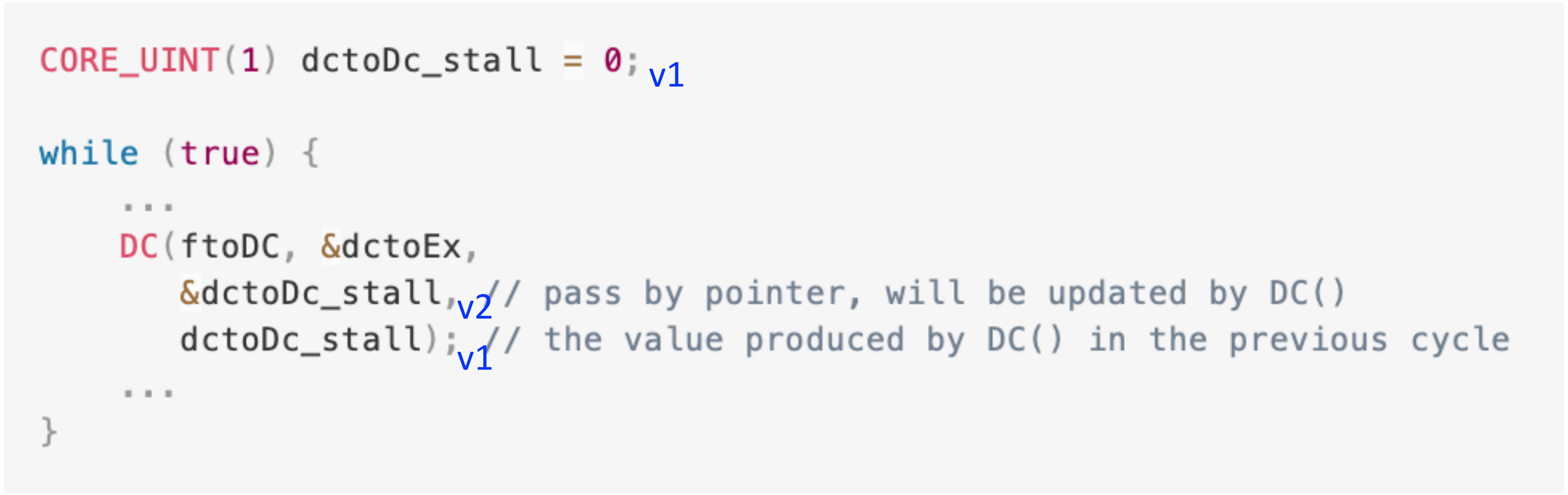
Self Signals

- A self signal can be implemented without introducing an additional variable

```
CORE_UINT(1) dctoDc_stall = 0; v0

while (true) {
    ...
    DC(ftoDC, &dctoEx,
       &dctoDc_stall, v1 // pass by pointer, will be updated by DC()
       dctoDc_stall); v0 // the value produced by DC() in the previous cycle
    ...
}
```

Self Signals

- A self signal can be implemented without introducing an additional variable



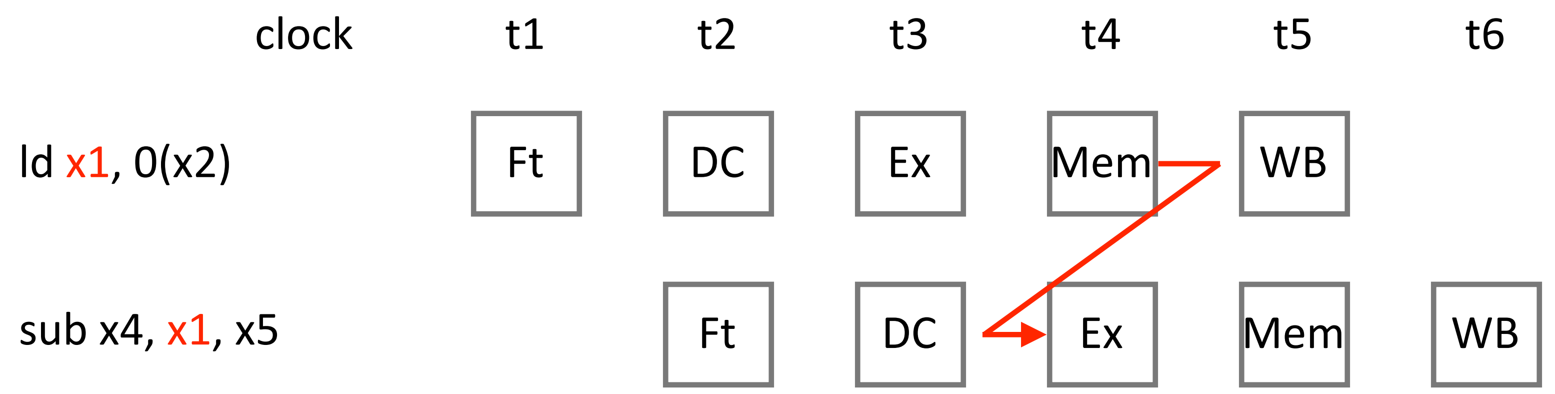
```
CORE_UINT(1) dctoDc_stall = 0; v1

while (true) {
    ...
    DC(ftoDC, &dctoEx,
       &dctoDc_stall, v2 // pass by pointer, will be updated by DC()
       dctoDc_stall); v1 // the value produced by DC() in the previous cycle
    ...
}
```

Outline

- Introduction to Comet
- Implementation
    - Pipeline stages and Pipeline registers
    - Signals
    - Stall, Flush, and Data Forwarding
    - Pipeline stage implementation
    - Performance Counters
- Workflow Demonstration

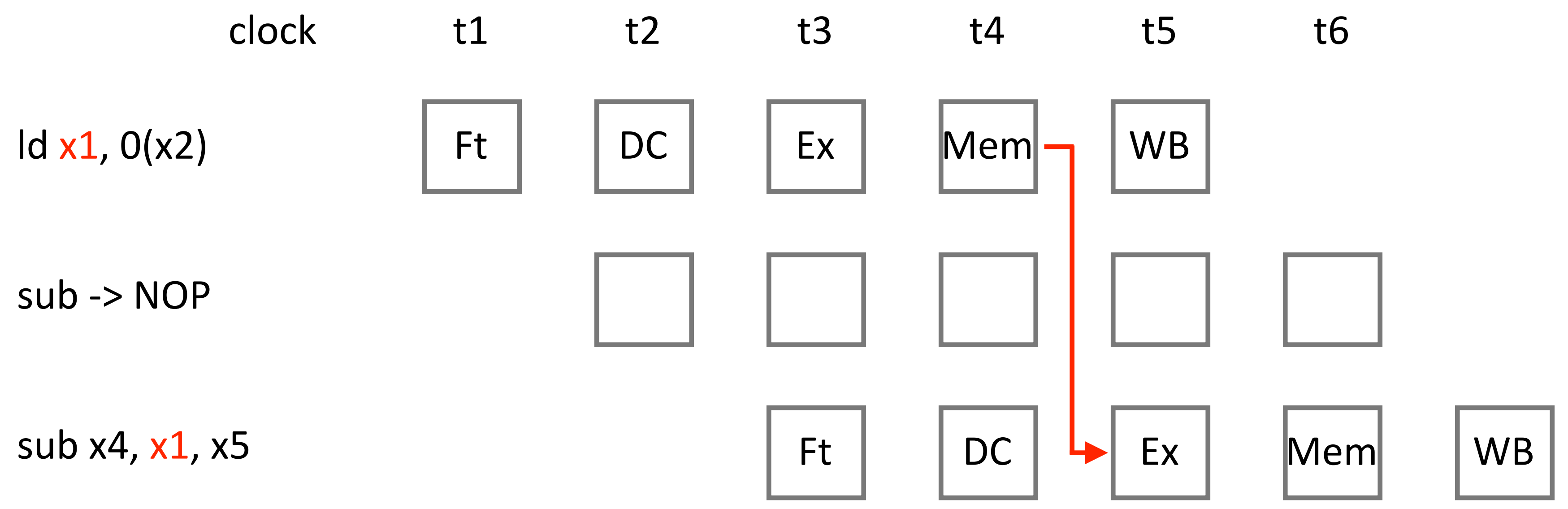
Stall

- Data hazards occur when an instruction needs the result of previous instruction, while the result of previous instruction is not yet available



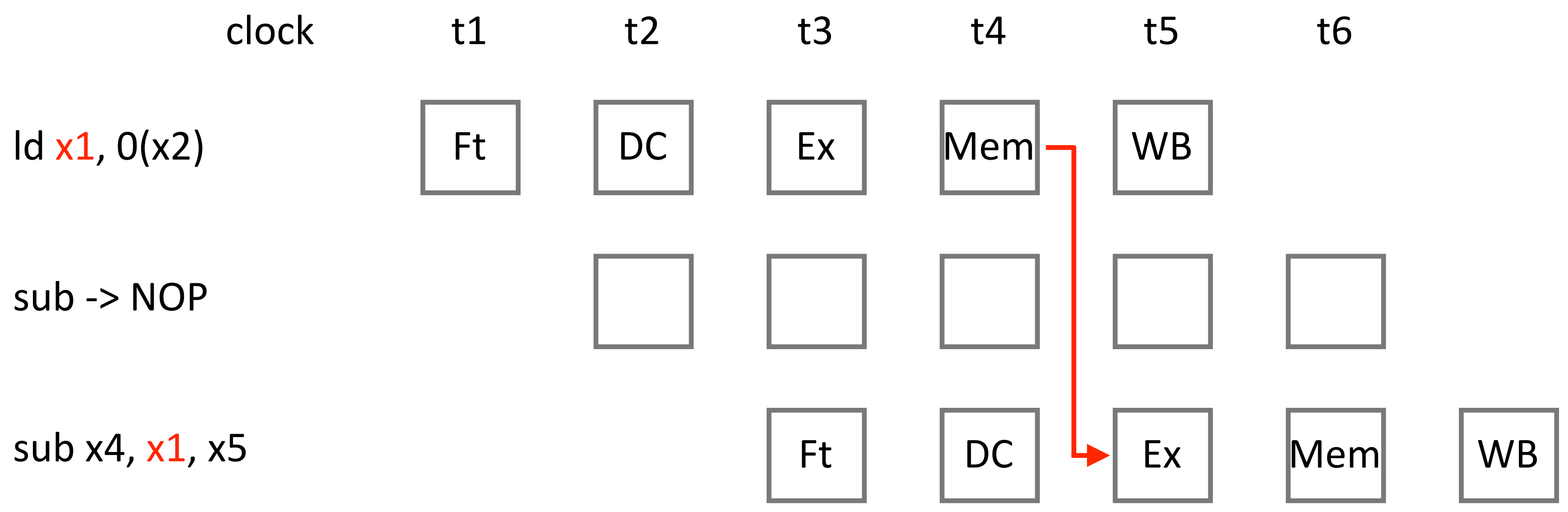
Stall

- Data hazards can be solved by inserting NOP instruction(s) between the two instruction, which is so-called Pipeline Stall



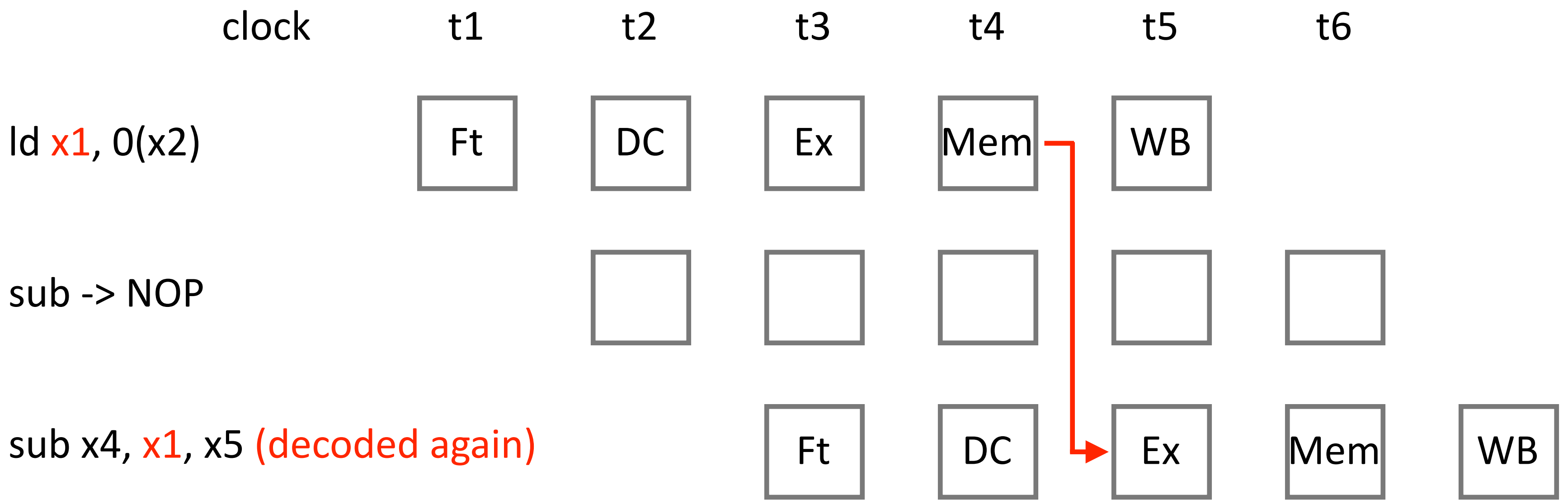
Stall

- The Ft stage should be informed not to update the pc until the hazard is solved



Stall

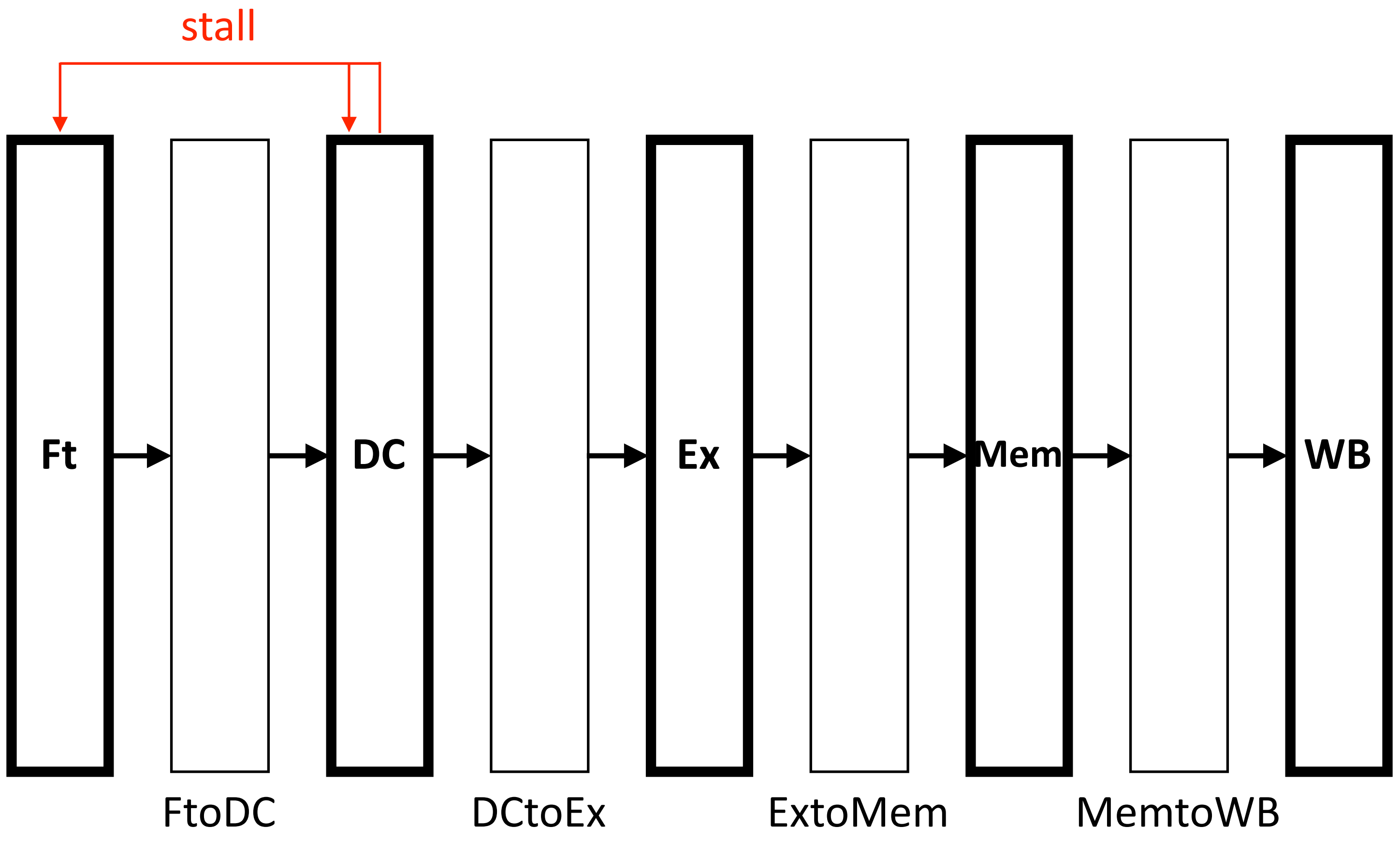
- The DC stage should replace the incoming instruction (and pc) with the previous instruction (and pc)



Stall

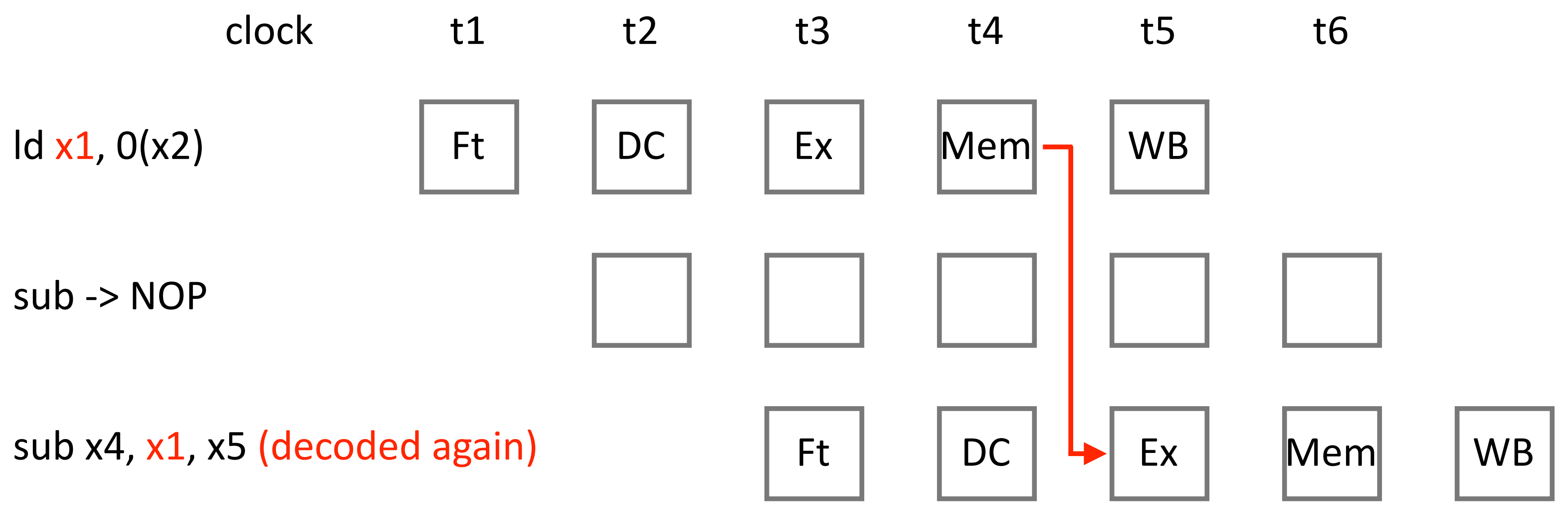
- Summary
  - Replace current instruction with NOP in DC
  - Decode current instruction in DC again in the next cycle
  - Keep pc from incrementing in Ft

Stall

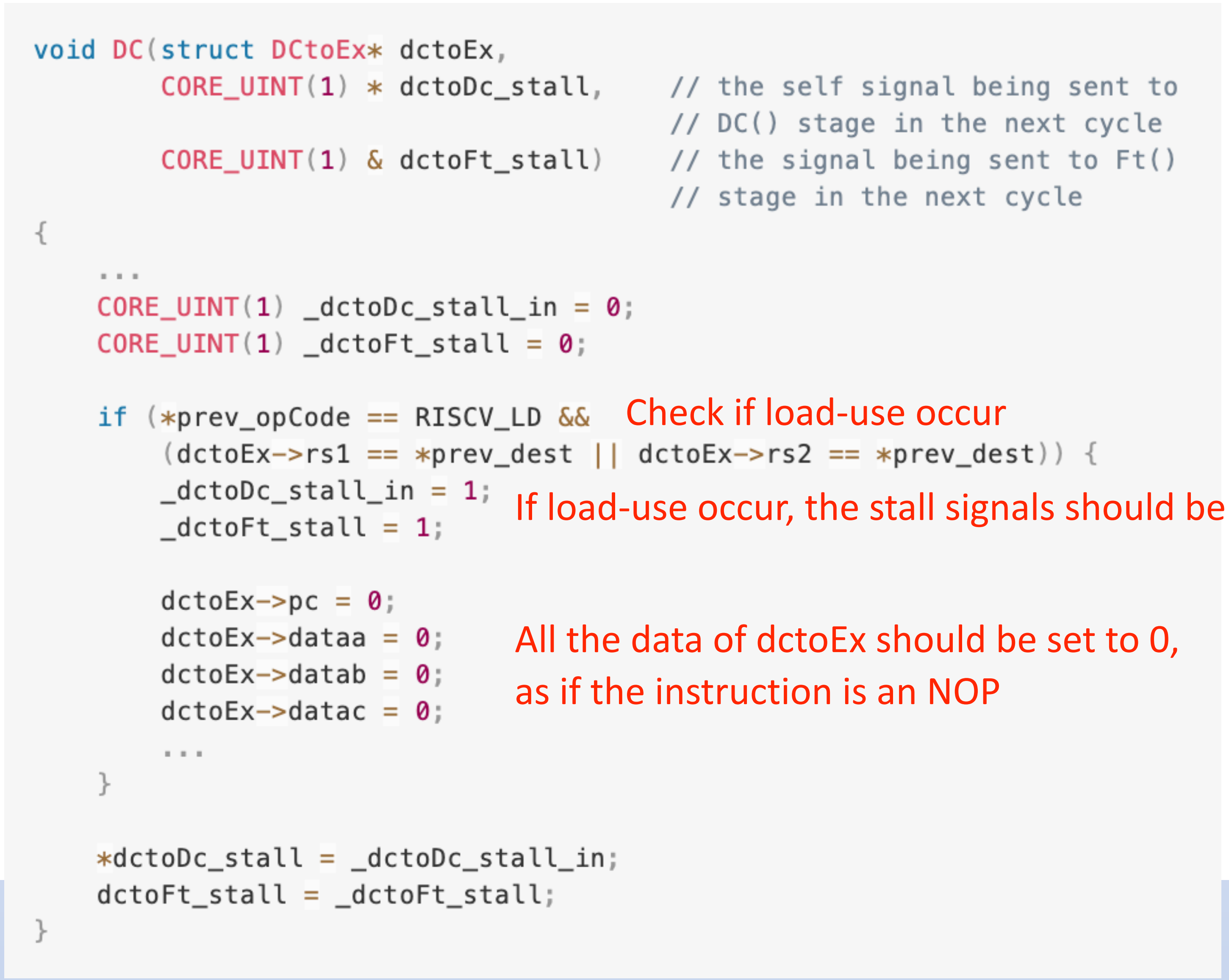


Stall

- In our basic design, the only possible cause of data hazard is the load-use condition



Stall - From DC



```
void DC(struct DCtoEx* dctoEx,
        CORE_UINT(1) * dctoDc_stall,    // the self signal being sent to
                                        // DC() stage in the next cycle
        CORE_UINT(1) & dctoFt_stall)    // the signal being sent to Ft()
                                        // stage in the next cycle
{
    ...
    CORE_UINT(1) _dctoDc_stall_in = 0;
    CORE_UINT(1) _dctoFt_stall = 0;

    if (*prev_opCode == RISCV_LD &&
        (dctoEx->rs1 == *prev_dest || dctoEx->rs2 == *prev_dest)) {
        _dctoDc_stall_in = 1;
        _dctoFt_stall = 1;

        dctoEx->pc = 0;
        dctoEx->dataa = 0;
        dctoEx->datab = 0;
        dctoEx->datac = 0;
        ...
    }

    *dctoDc_stall = _dctoDc_stall_in;
    dctoFt_stall = _dctoFt_stall;
}
```

Check if load-use occur

If load-use occur, the stall signals should be set to 1

All the data of dctoEx should be set to 0,
as if the instruction is an NOP

Stall - to Ft

```
void Ft(CORE_UINT(32) * pc,
        CORE_INT(32) ins_memory[MEM_SIZE / 4],
        struct FtoDC* ftoDC,
        CORE_UINT(1) _dctoFt_stall)
{
    CORE_UINT(32) next_pc = *pc;

    if (!_dctoFt_stall) {
        next_pc += 4;
        *pc = next_pc;
        (ftoDC->instruction).SET_SLC(0, ins_memory[(*pc & 0x0FFFF) / 4]);
        ftoDC->pc = *pc;
    }
}
```

If a stall occur, the pc should not be added

Stall - to DC

```
void DC(struct FtoDC ftoDC,
        CORE_UINT(32) * prev_instruction,
        CORE_UINT(1) _dctoDc_stall_out) // the self signal produced
                                        // in the previous cycle
{
    if (_dctoDc_stall_out) {
        ftoDC.instruction = *prev_instruction;
        ftoDC.pc = *prev_pc;
    }
    ...
}
```

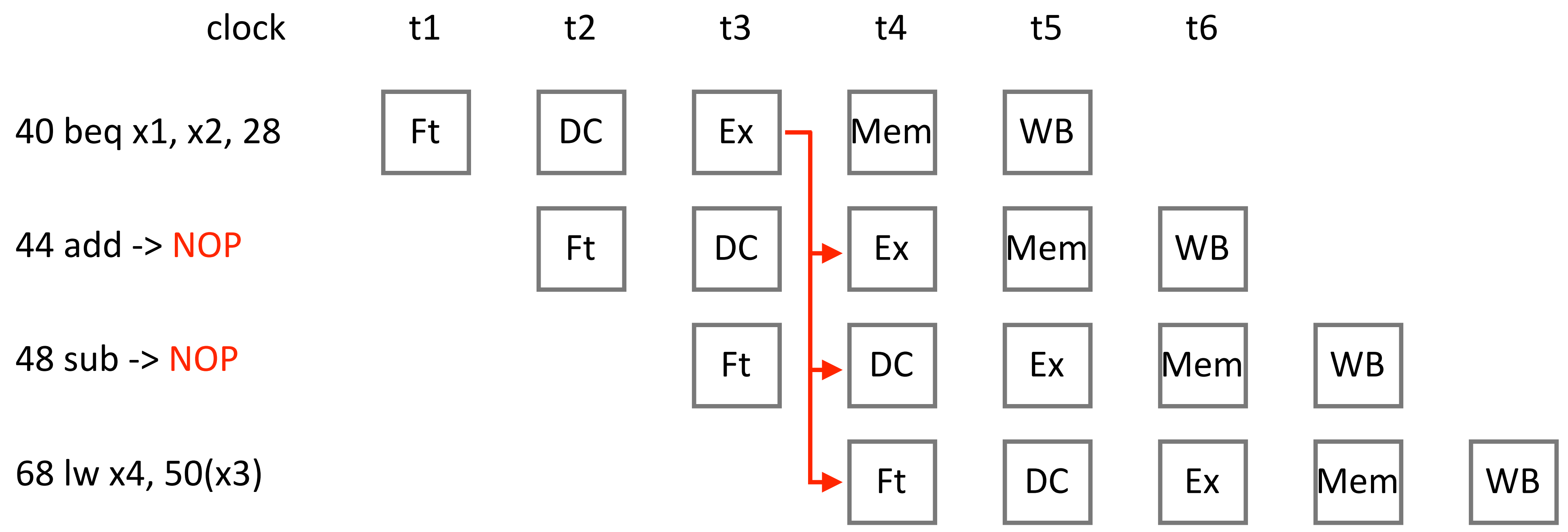
The previous instruction should be processed again,
since the instruction was treated as an NOP in the previous cycle

Flush

- Control hazards occur when the pipeline makes wrong decisions on branch prediction
- For example, if a branch is predicted non-taken, but the result of Ex stage implies that the branch should be taken, then the wrong instructions should be discard

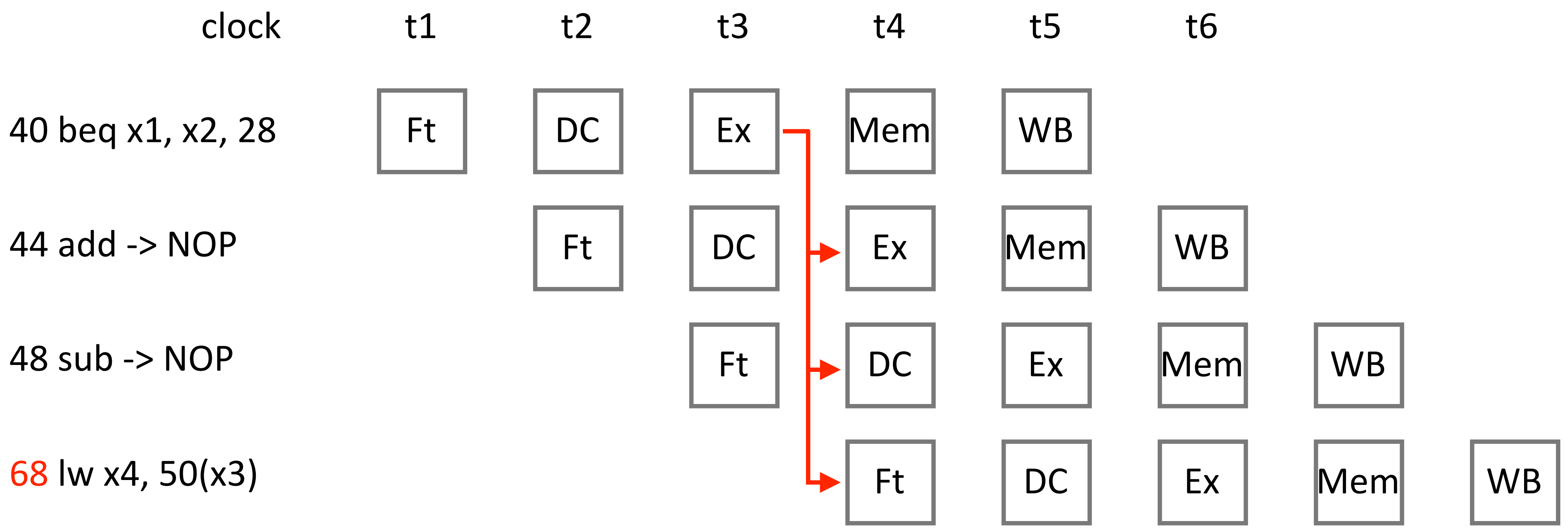
Flush

- Similarly to data hazards, control hazards can be solved by replacing the wrong instructions with NOPs, which is so-called Pipeline Flush



Flush

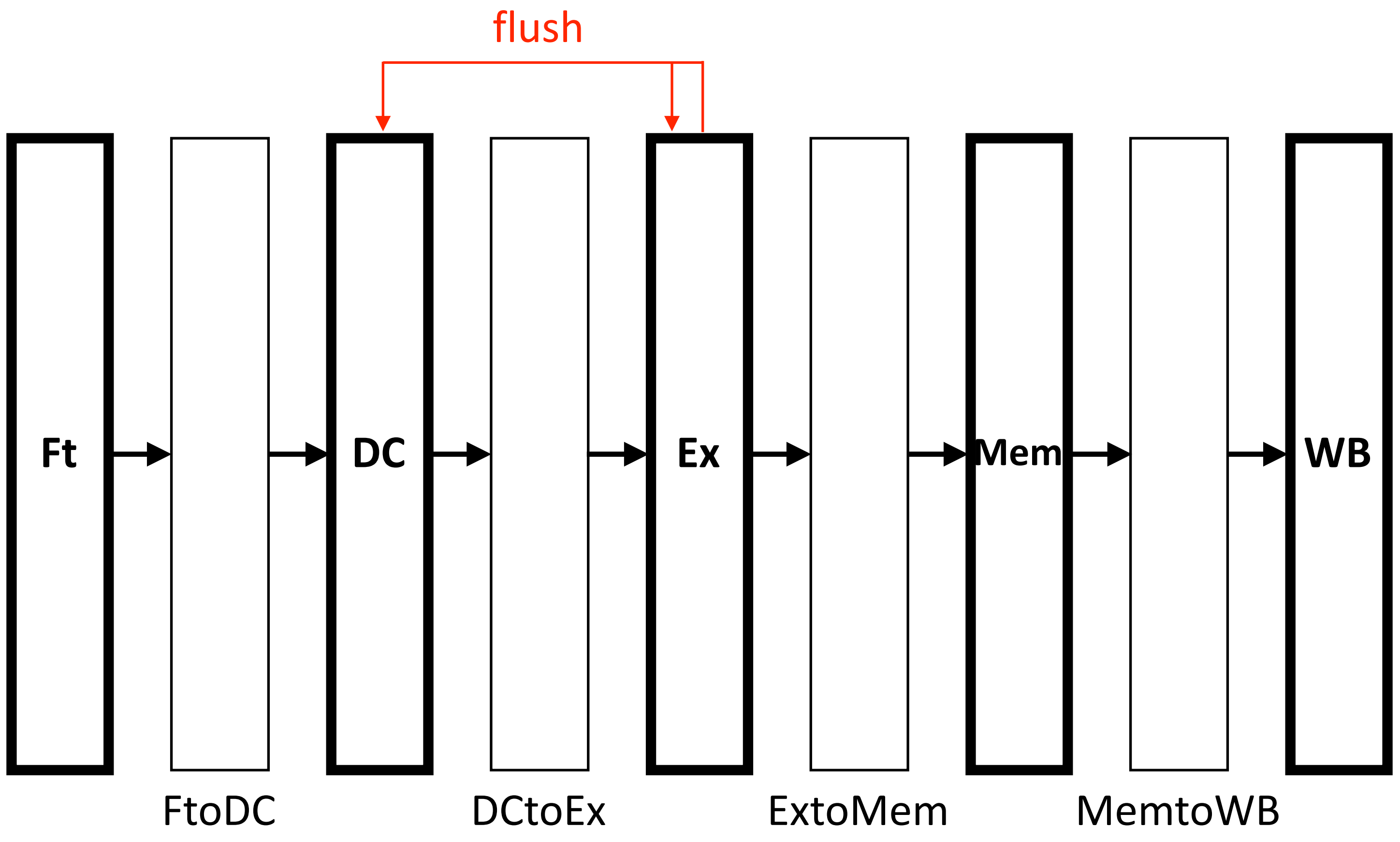
- Moreover, the pc in Ft stage should be updated to the address of the correct branch, so that the correct instruction can be fetched



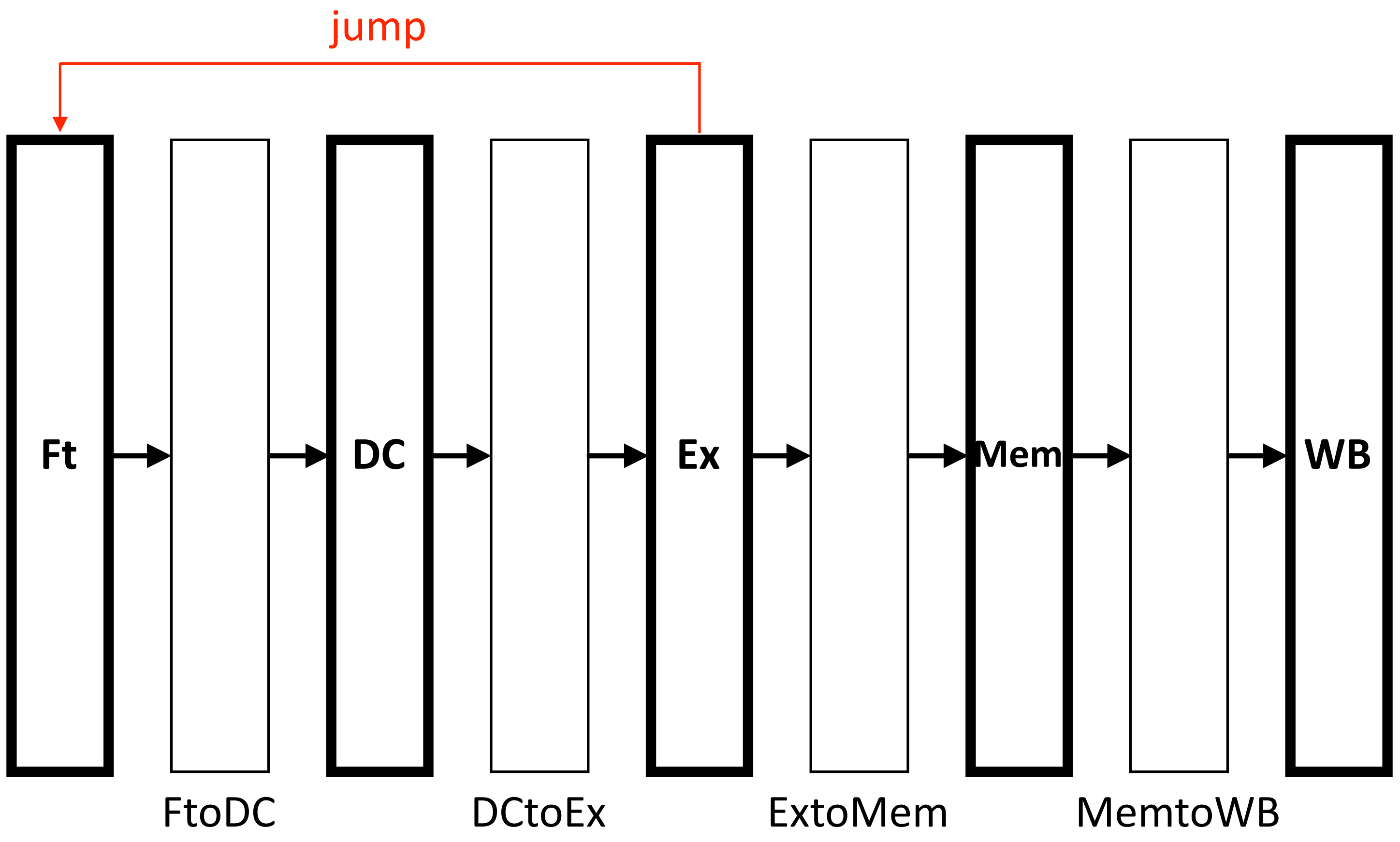
Flush

- Summary
    - Replace the next instructions in DC and Ex with NOPs
    - Update pc in Ft

Flush



Flush



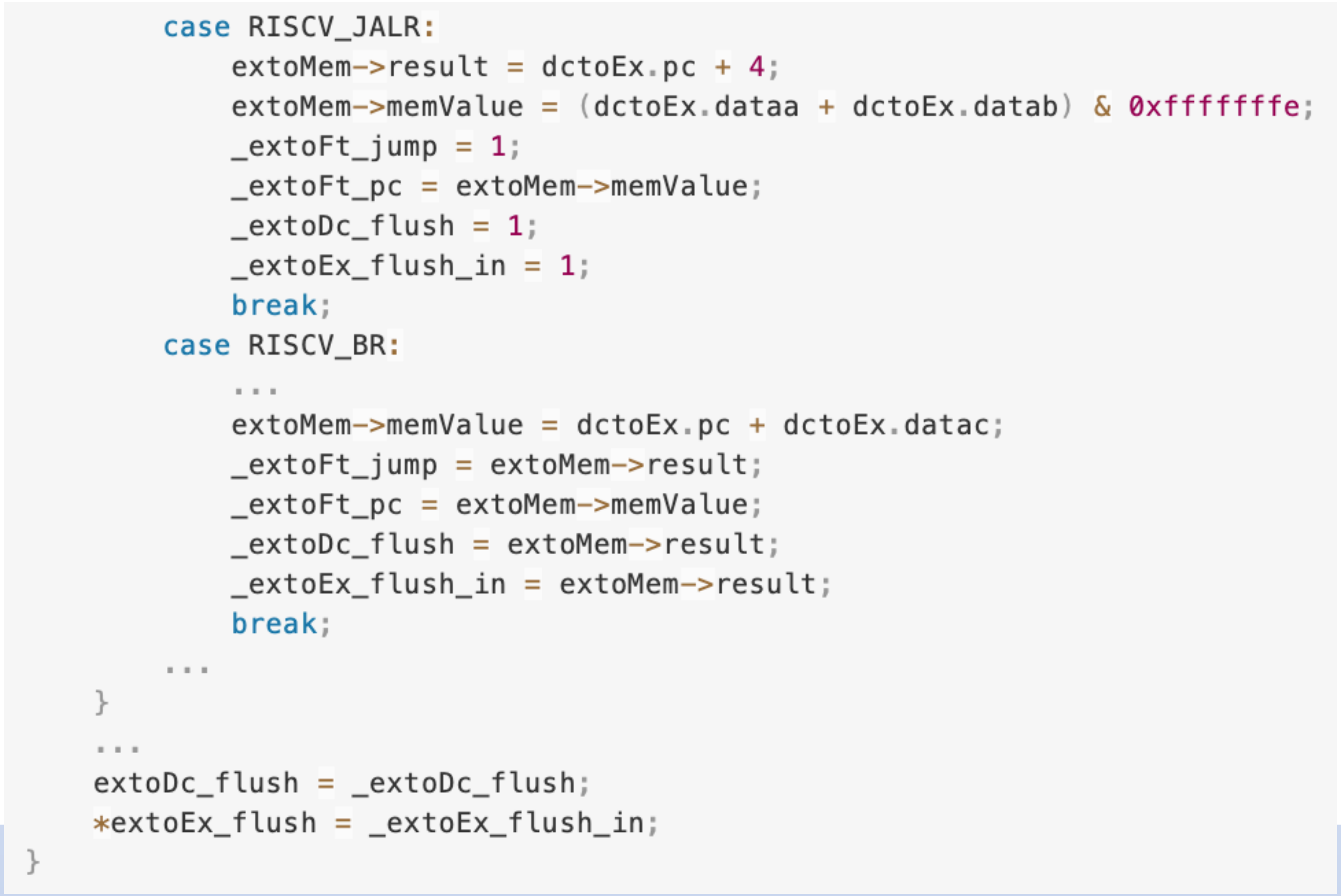
Flush

- In our basic design, the possible causes of branching are
  - Jump instructions (JAL and JR), and
  - Branch mispredictions
- More specifically, the branch are always predicted non-taken, so the control hazard may occur if the branch should be taken

Flush - from Ex

```
void Ex(struct DCtoEx dctoEx,
        struct ExtoMem* extoMem,
        CORE_UINT(1) & extoFt_jump,  // to Ft() stage
        CORE_UINT(32) & extoFt_pc,   // to Ft() stage
        CORE_UINT(1) & extoDc_flush, // to DC() stage
        CORE_UINT(1) * extoEx_flush) // self signal being sent to Ex()
                                     // in the next cycle
{
    CORE_UINT(1) _extoDc_flush = 0;
    CORE_UINT(1) _extoEx_flush_in = 0;
    ...
    switch (dctoEx.opCode) {
        case RISCV_JAL:
            extoMem->result = dctoEx.pc + 4;
            extoMem->memValue = dctoEx.pc + dctoEx.datab;
            _extoFt_jump = 1;
            _extoFt_pc = extoMem->memValue;
            _extoDc_flush = 1;
            _extoEx_flush_in = 1;
            break;
```

Flush - from Ex

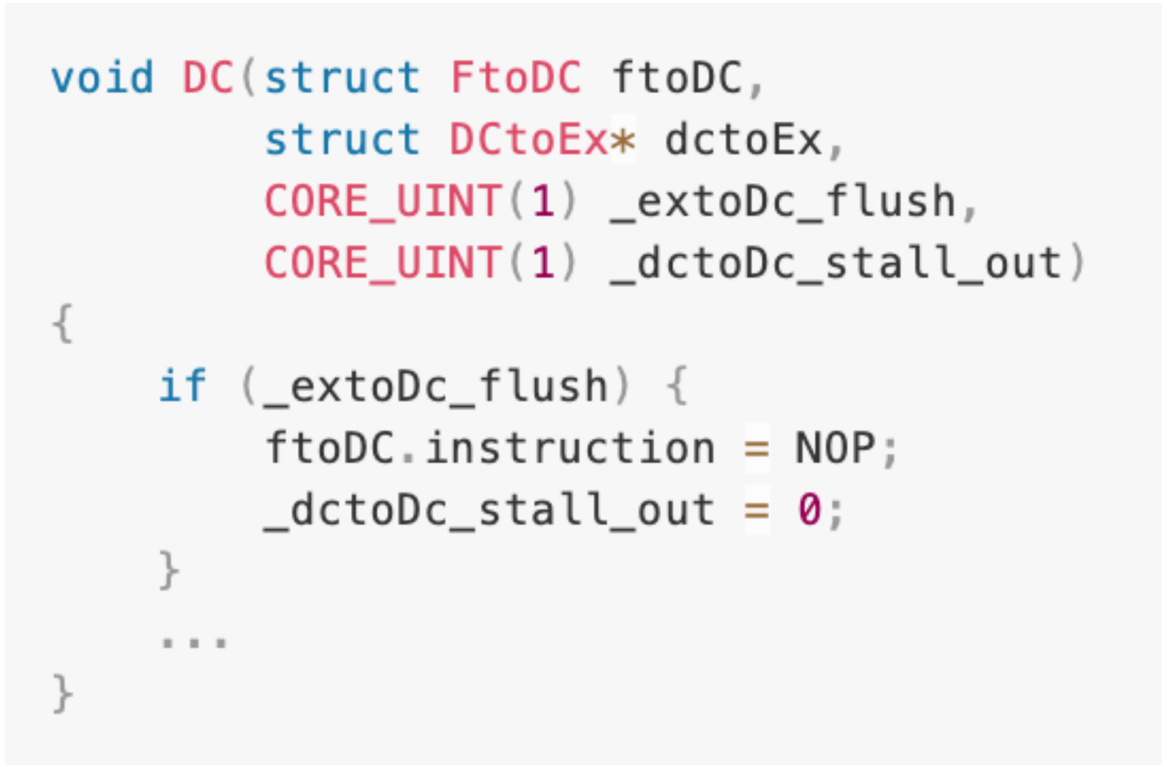


```
        case RISCV_JALR:
            extoMem->result = dctoEx.pc + 4;
            extoMem->memValue = (dctoEx.dataa + dctoEx.datab) & 0xfffffffe;
            _extoFt_jump = 1;
            _extoFt_pc = extoMem->memValue;
            _extoDc_flush = 1;
            _extoEx_flush_in = 1;
            break;
        case RISCV_BR:
            ...
            extoMem->memValue = dctoEx.pc + dctoEx.datac;
            _extoFt_jump = extoMem->result;
            _extoFt_pc = extoMem->memValue;
            _extoDc_flush = extoMem->result;
            _extoEx_flush_in = extoMem->result;
            break;
        ...
    }
    ...
    extoDc_flush = _extoDc_flush;
    *extoEx_flush = _extoEx_flush_in;
}
```

Flush - to Ft

```
void Ft(CORE_UINT(32) * pc,
        CORE_INT(32) ins_memory[MEM_SIZE / 4],
        struct FtoDC* ftoDC,
        CORE_UINT(1) _extoFt_jump,
        CORE_UINT(32) _extoFt_pc,
        CORE_UINT(1) _dctoFt_stall)
{
    CORE_UINT(32) next_pc = *pc;

    if (!_dctoFt_stall) next_pc += 4;
    *pc = _extoFt_jump ? _extoFt_pc : next_pc;
    if (!_dctoFt_stall) {
        (ftoDC->instruction).SET_SLC(0, ins_memory[(*pc & 0x0FFFF) / 4]);
        ftoDC->pc = *pc;
    }
}
```

Flush - to DC



```
void DC(struct FtoDC ftoDC,
        struct DCtoEx* dctoEx,
        CORE_UINT(1) _extoDc_flush,
        CORE_UINT(1) _dctoDc_stall_out)
{
    if (_extoDc_flush) {
        ftoDC.instruction = NOP;
        _dctoDc_stall_out = 0;
    }
    ...
}
```

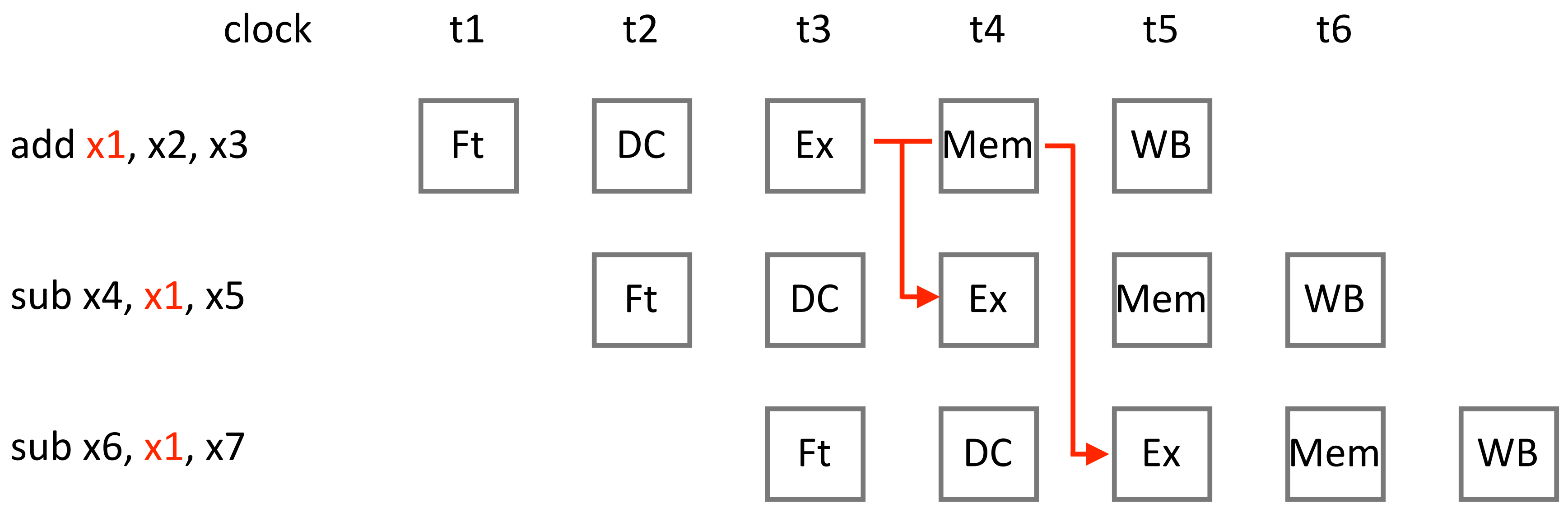
Flush - to Ex

```
void Ex(struct DCtoEx dctoEx,
        struct ExtoMem* extoMem,
        CORE_UINT(1) _extoEx_flush_out)
{
    if (_extoEx_flush_out) {
        dctoEx.dataa = 0;
        dctoEx.datab = 0;
        dctoEx.dest = 0;
        dctoEx.opCode = RISCV_OPI;
        dctoEx.memValue = 0;
        dctoEx.funct3 = RISCV_OPI_ADDI;
    }
    ...
}
```

addi x0, x0, 0

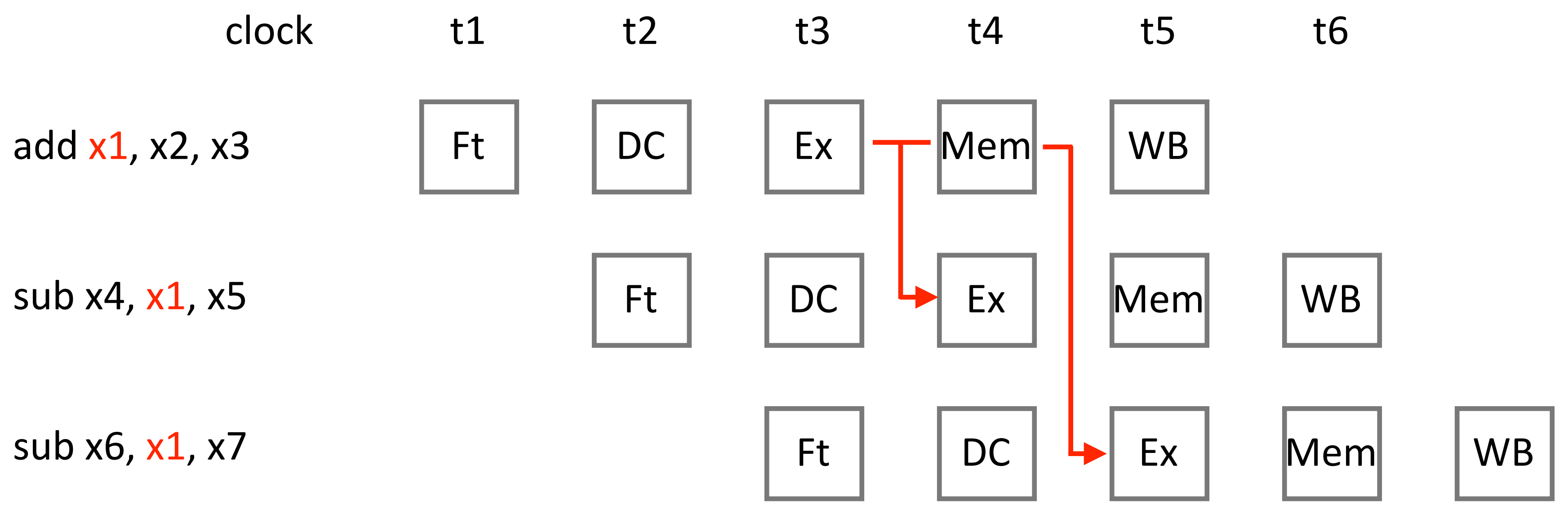
Data Forwarding

- Some of the data hazards can be solved by passing the result value back to the stage which needs the value



Data Forwarding

- The result of the Ex and Mem stages will be forwarded every cycle, so that the Ex stage can determine if the forwarded value should be taken



Data Forwarding

- Summary
  - Forward the result of Ex and Mem back to Ex
  - Determine if the forwarded value should be taken in Ex

Data Forwarding - from Ex

```
void Ex(struct DCtoEx dctoEx,
        struct ExtoMem* extoMem,
        CORE_UINT(5) & extoEx_fwd_dest,
        CORE_INT(32) & extoEx_fwd_value,
        CORE_UINT(1) & extoEx_fwd_load)
{
    ...
    // Forward the data only if write-back is enabled
    CORE_UINT(5) _extoEx_fwd_dest_in =
        extoMem->WBena ? extoMem->dest : 0;
    CORE_INT(32) _extoEx_fwd_value_in =
        extoMem->WBena ? extoMem->result : 0;
    CORE_UINT(1) _extoEx_fwd_load_in =
        extoMem->WBena ? (dctoEx.opCode == RISCV_ST) : 0;

    extoEx_fwd_dest = _extoEx_fwd_dest_in;   // the rd of current instruction
    extoEx_fwd_value = _extoEx_fwd_value_in; // the result of execution
    extoEx_fwd_load = _extoEx_fwd_load_in;   // is this a load instruction?
}
```

Data Forwarding - from Mem

```
void do_Mem(struct ExtoMem extoMem,
            struct MemtoWB* memtoWB,
            CORE_UINT(5) & memtoEx_fwd_dest,
            CORE_INT(32) & memtoEx_fwd_value)
{
    ...
    CORE_UINT(5) _memtoEx_fwd_dest =
        memtoWB->WBena ? memtoWB->dest : 0;
    CORE_INT(32) _memtoEx_fwd_value =
        memtoWB->WBena ? memtoWB->result : 0;

    memtoEx_fwd_dest = _memtoEx_fwd_dest;
    memtoEx_fwd_value = _memtoEx_fwd_value;
}
```

Data Forwarding

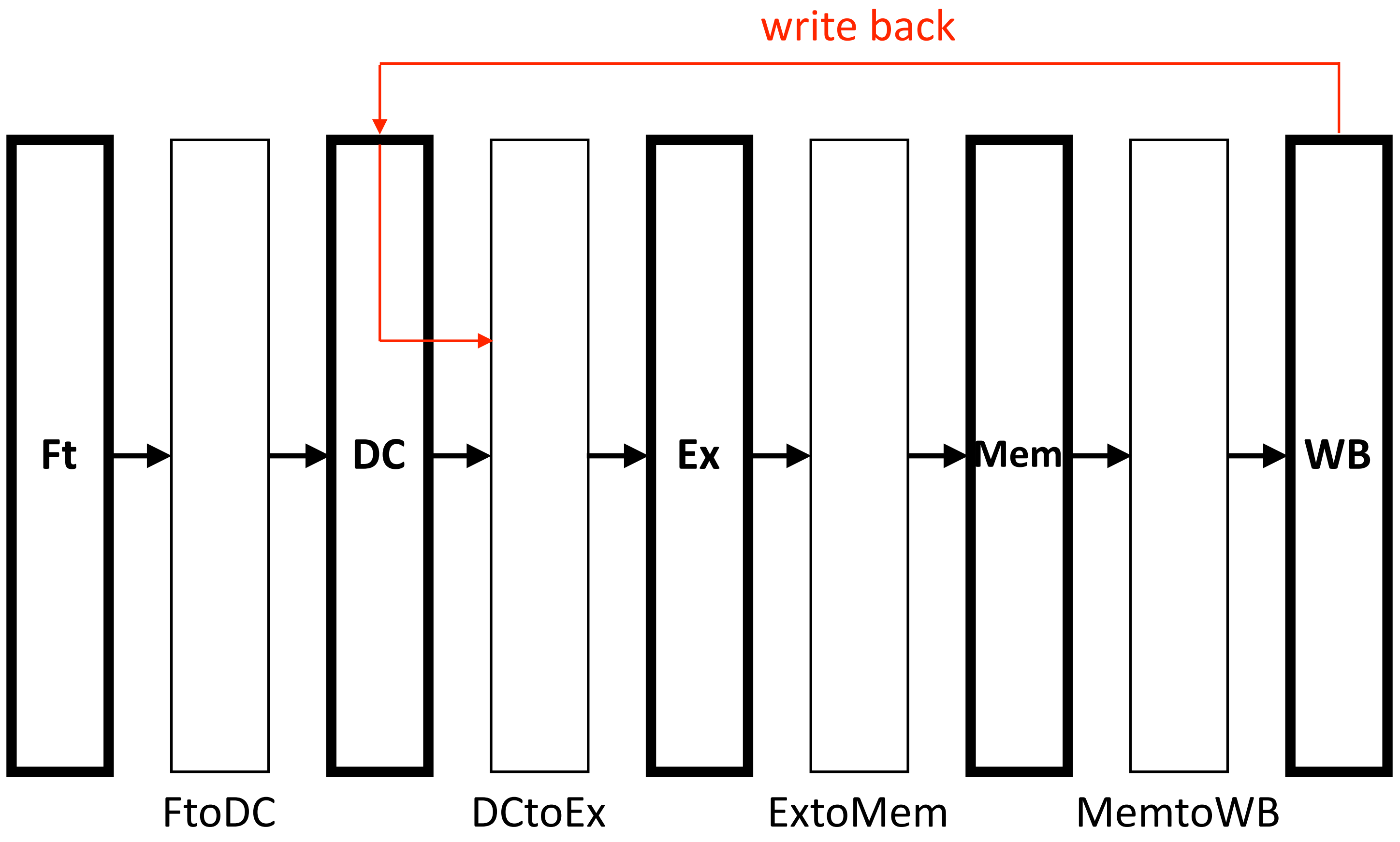
- At the begin of the Ex stage, it checks if the forwarded rd matches the dctoEx.rs1 or dctoEx.rs2
- It is possible that both rd from the Ex stage and the Mem stage match
- In this case, the latest result should be taken

Data Forwarding

- A RISC-V pipelined processor designed in RTL may only have to handle the data forwarding from the Ex stage and the Mem stage
- However, different from the implementation of RTL, the data forwarding from the WB stage is required in our HLS implementation

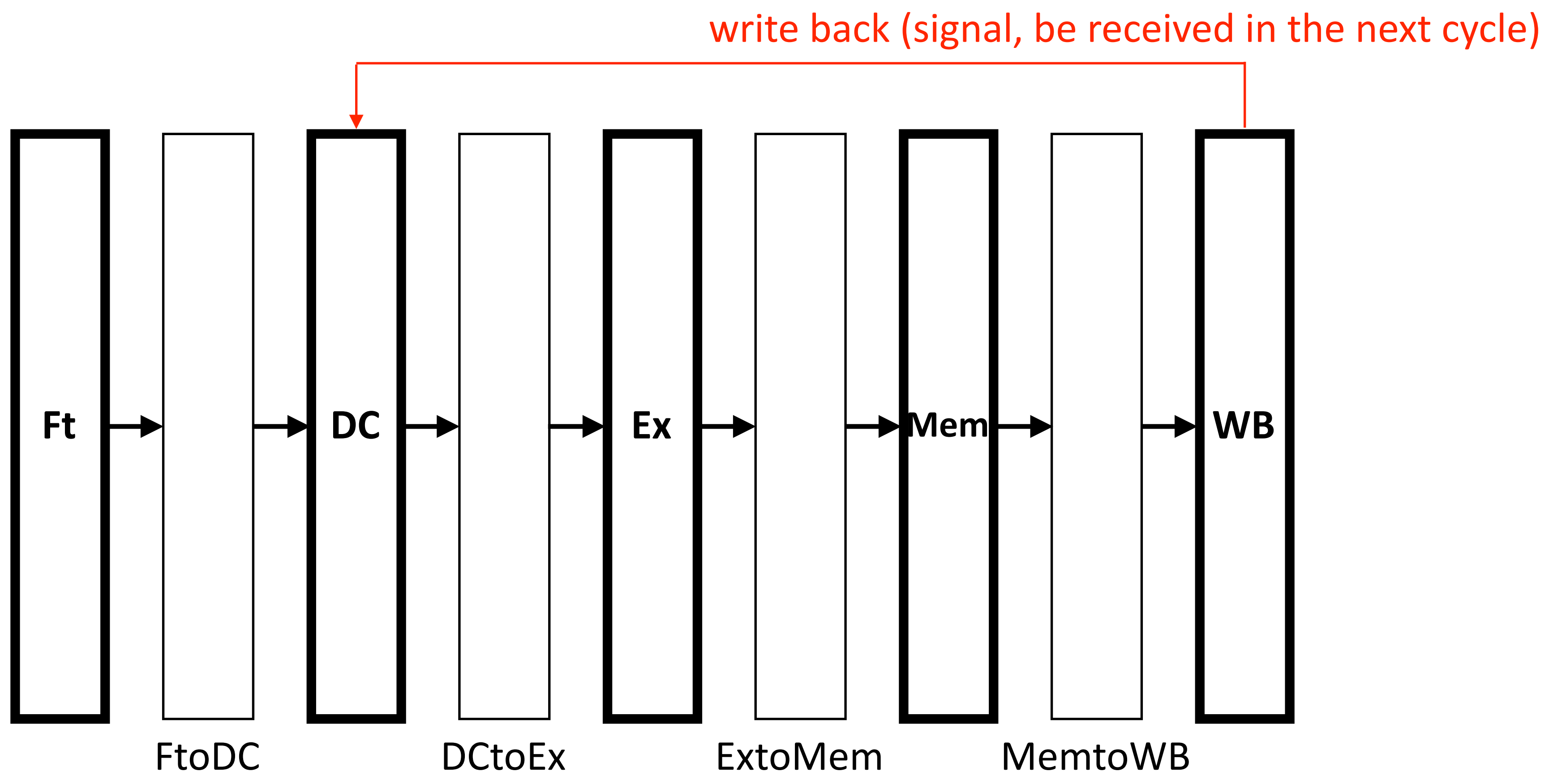
Data Forwarding

RTL



Data Forwarding

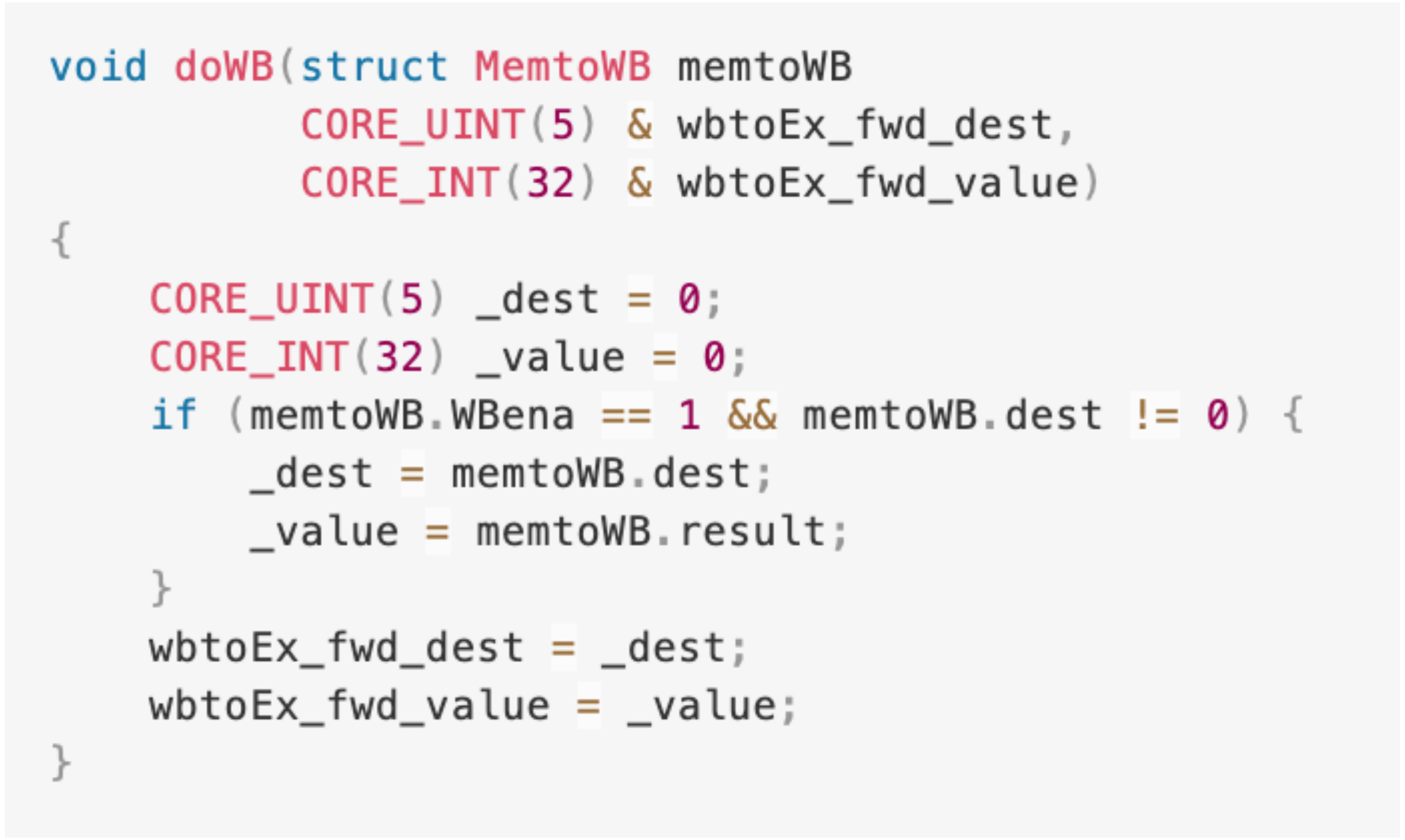
HLS



Data Forwarding

- Summary
  - Forward the result of Ex and Mem and WB back to Ex
  - Determine if the forwarded value should be taken in Ex

Data Forwarding - from WB



```
void doWB(struct MemtoWB memtoWB
          CORE_UINT(5) & wbtoEx_fwd_dest,
          CORE_INT(32) & wbtoEx_fwd_value)
{
    CORE_UINT(5) _dest = 0;
    CORE_INT(32) _value = 0;
    if (memtoWB.WBena == 1 && memtoWB.dest != 0) {
        _dest = memtoWB.dest;
        _value = memtoWB.result;
    }
    wbtoEx_fwd_dest = _dest;
    wbtoEx_fwd_value = _value;
}
```

Data Forwarding

- At the begin of the Ex stage, it checks if the forwarded rd matches the dctoEx.rs1 or dctoEx.rs2
- It is possible that all rd from the Ex stage, the Mem stage and the WB stage match
- In this case, the latest result should be taken

Data Forwarding - to Ex

```
if (dctoEx.rs1 != 0) {
    if (dctoEx.rs1 == _extoEx_fwd_dest_out &&
        !_extoEx_fwd_load_out) {
        dctoEx.dataa = _extoEx_fwd_value_out;
    } else if (dctoEx.rs1 == _memtoEx_fwd_dest_out) {
        dctoEx.dataa = _memtoEx_fwd_value_out;
    } else if (dctoEx.rs1 == _wbtoEx_fwd_dest_out) {
        dctoEx.dataa = _wbtoEx_fwd_value_out;
    }
}
```

Outline

- Introduction to Comet
- Implementation
    - Pipeline stages and Pipeline registers
    - Signals
    - Stall, Flush, and Data Forwarding
    - Pipeline stage implementation
    - Performance Counters
- Workflow Demonstration

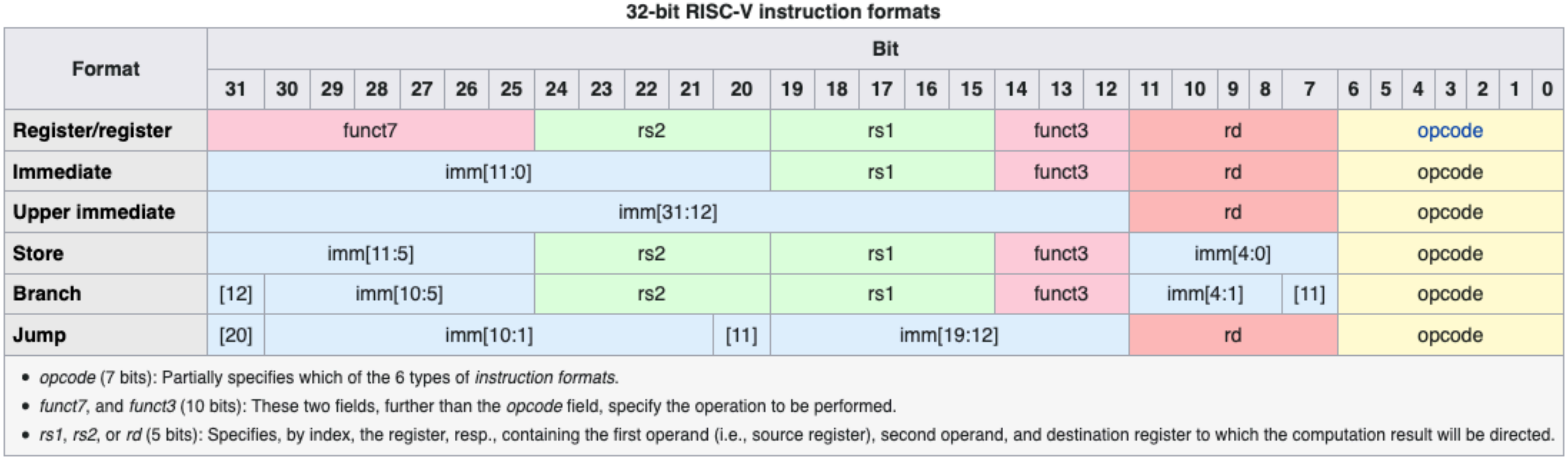
Ft Stage

```
CORE_UINT(32) next_pc = *pc;
if (!_dctoFt_stall) next_pc += 4;

*pc = _extoFt_jump ? _extoFt_pc : next_pc;

if (!_dctoFt_stall) {
    (ftoDC->instruction).SET_SLC(0, ins_memory[(*pc & 0x0FFFF) / 4]);
    ftoDC->pc = *pc;
}
```

DC Stage

- In RISC-V ISA, different types of instructions have different formats
- For example, R-type instructions have func7 and rs2 fields, while I-type instructions do not, and the corresponding field in I-type instructions represents immediate
- DC function simply decodes all fields, and uses only some of them

DC Stage

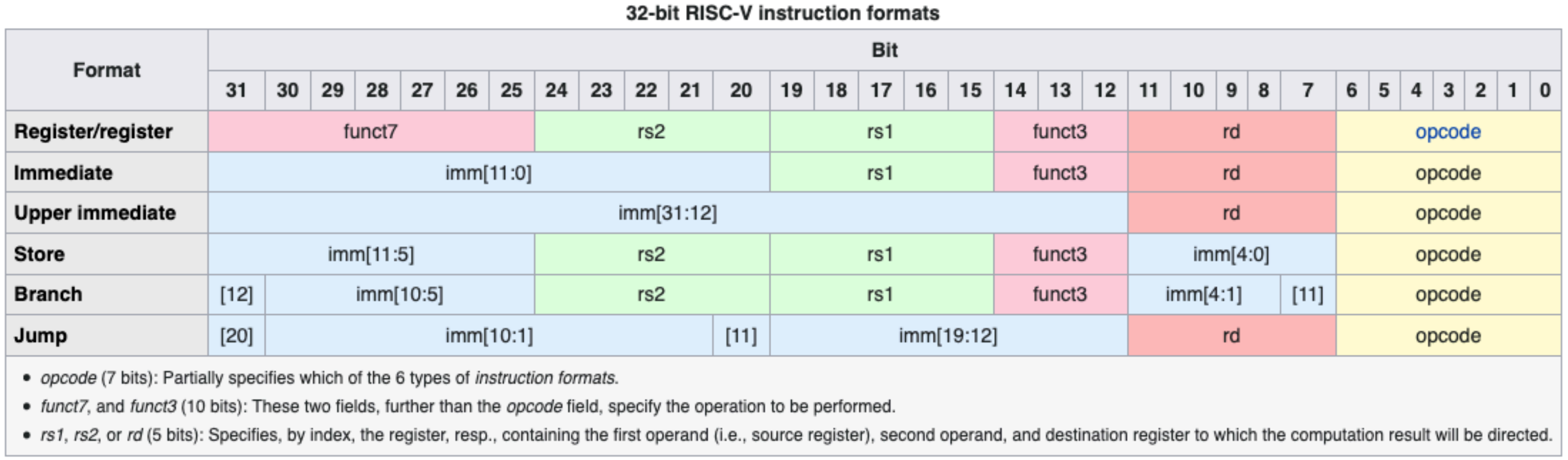


32-bit RISC-V instruction formats

Format	Bit																															
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Register/register	funct7							rs2					rs1					funct3			rd					opcode						
Immediate	imm[11:0]												rs1					funct3			rd					opcode						
Upper immediate	imm[31:12]																				rd					opcode						
Store	imm[11:5]							rs2					rs1					funct3			imm[4:0]					opcode						
Branch	[12]	imm[10:5]						rs2					rs1					funct3			imm[4:1]				[11]	opcode						
Jump	[20]	imm[10:1]										[11]	imm[19:12]								rd					opcode						

- *opcode* (7 bits): Partially specifies which of the 6 types of *instruction formats*.
- *funct7*, and *funct3* (10 bits): These two fields, further than the *opcode* field, specify the operation to be performed.
- *rs1*, *rs2*, or *rd* (5 bits): Specifies, by index, the register, resp., containing the first operand (i.e., source register), second operand, and destination register to which the computation result will be directed.

DC Stage



32-bit RISC-V instruction formats

Format	Bit																															
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Register/register	funct7							rs2					rs1					funct3			rd					opcode						
Immediate	imm[11:0]												rs1					funct3			rd					opcode						
Upper immediate	imm[31:12]																				rd					opcode						
Store	imm[11:5]							rs2					rs1					funct3			imm[4:0]					opcode						
Branch	[12]	imm[10:5]						rs2					rs1					funct3			imm[4:1]				[11]	opcode						
Jump	[20]	imm[10:1]										[11]	imm[19:12]								rd					opcode						

- *opcode* (7 bits): Partially specifies which of the 6 types of *instruction formats*.
- *funct7*, and *funct3* (10 bits): These two fields, further than the *opcode* field, specify the operation to be performed.
- *rs1*, *rs2*, or *rd* (5 bits): Specifies, by index, the register, resp., containing the first operand (i.e., source register), second operand, and destination register to which the computation result will be directed.

```
CORE_UINT(5) rs1 = ftoDC.instruction.SLC(5, 15);
CORE_UINT(5) rs2 = ftoDC.instruction.SLC(5, 20);
CORE_UINT(5) rd = ftoDC.instruction.SLC(5, 7);
CORE_UINT(7) opcode = ftoDC.instruction.SLC(7, 0);
CORE_UINT(7) funct7 = ftoDC.instruction.SLC(7, 25);
CORE_UINT(7) funct3 = ftoDC.instruction.SLC(3, 12);
...
```

DC Stage

- DC function reads the values from registers according to the rs fields
- The write-back of register values (which is implemented in signal from WB to DC) should also be handled in DC
- Since a register can possibly be read and updated in the same cycle, we should avoid reading from and writing to the register in the same cycle

DC Stage

- ARRAY_PARTITION pragma should be added to remove the dependencies between different registers

```
#pragma HLS ARRAY_PARTITION variable = REG complete

CORE_INT(32) reg_rs1 =
    (rs1 != 0 && rs1 == _wbtoDc_reg_dest) ? _wbtoDc_reg_value : REG[rs1];
CORE_INT(32) reg_rs2 =
    (rs2 != 0 && rs2 == _wbtoDc_reg_dest) ? _wbtoDc_reg_value : REG[rs2];
if (_wbtoDc_reg_dest != 0) {
    REG[_wbtoDc_reg_dest] = _wbtoDc_reg_value;
}
```

DC Stage

- Then, the DC function determines the operation type using the opcode field, and passes the corresponding fields to the pipeline register dctoEx
- The meaningless fields (such as rs2 in I-type) will remain 0 in dctoEx, only the necessary fields are set

DC Stage

```
switch (opcode) {
   ...
    case RISCV_BR:
        dctoEx->rs2 = rs2;
        dctoEx->datac = imm13_signed;
        dctoEx->dest = 0;
        break;
    case RISCV_LD:
        dctoEx->dest = rd;
        dctoEx->memValue = imm12_I_signed;
        break;
    case RISCV_ST:
        dctoEx->rs2 = rs2;
        dctoEx->memValue = store_imm;
        dctoEx->dest = 0;
        break;
    case RISCV_OPI:
        dctoEx->dest = rd;
        dctoEx->memValue = imm12_I_signed;
        dctoEx->datab = imm12_I;
        break;
    ...
}
```

Ex Stage

- Ex function determines if write-back should be enabled
- In fact, in our instruction set (RV32I + MUL), almost all instructions write the result back to the registers, except for branch instructions and store instructions

Ex Stage

```
if ((extoMem->opCode != RISCV_BR) && (extoMem->opCode != RISCV_ST)) {
    extoMem->WBena = 1;
} else {
    extoMem->WBena = 0;
}
```

Ex Stage

- According to the opcode, and possibly funct3 and funct7, in the pipeline register dctoEx, corresponding operations are performed
- Also, some signals may be set according to the operation type, or the operation result

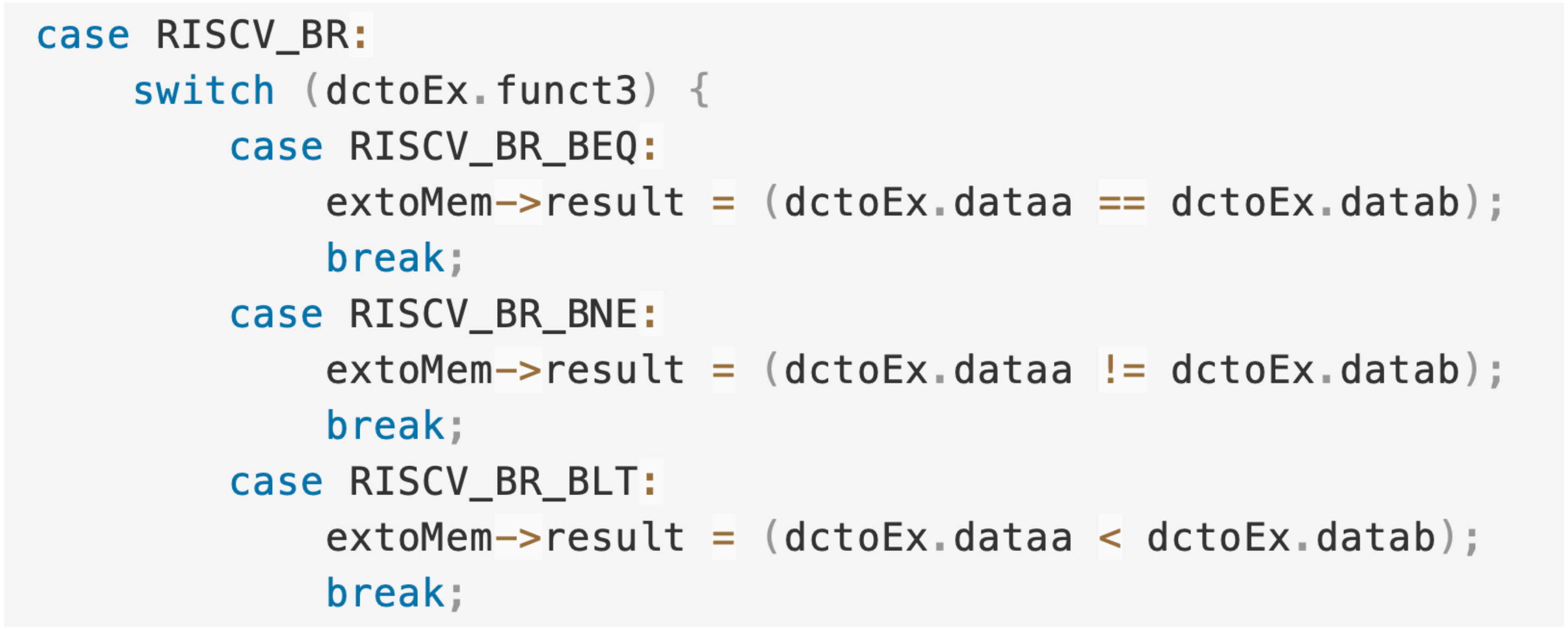
Ex Stage

- For example, if a jump instruction is executed, the flush signal should be set anyway

```
switch (dctoEx.opCode) {
    ...
    case RISCV_JAL:
        extoMem->result = dctoEx.pc + 4;
        extoMem->memValue = dctoEx.pc + dctoEx.datab;
        _extoFt_jump = 1;
        _extoFt_pc = extoMem->memValue;
        _extoDc_flush = 1;
        _extoEx_flush_in = 1;
        break;
```

Ex Stage

- If a conditional branch instruction is executed, the flush signal should be set if the condition is met



```
case RISCV_BR:
    switch (dctoEx.funct3) {
        case RISCV_BR_BEQ:
            extoMem->result = (dctoEx.dataa == dctoEx.datab);
            break;
        case RISCV_BR_BNE:
            extoMem->result = (dctoEx.dataa != dctoEx.datab);
            break;
        case RISCV_BR_BLT:
            extoMem->result = (dctoEx.dataa < dctoEx.datab);
            break;
```

Ex Stage

```
extoMem->memValue = dctoEx.pc + dctoEx.datac;
_extoFt_jump = extoMem->result;
_extoFt_pc = extoMem->memValue;
_extoDc_flush = extoMem->result;
break;
```

Mem Stage

- Similar to the Ex stage, the Mem stage uses a switch-case on opcode to determine which kind of operation is needed
- The difference is that the Mem stage only handles load instructions and store instructions, performing corresponding access to the data memory

Mem Stage

```
switch (extoMem.opCode) {
    ...
    case RISCV_LD:
        ...
        memtoWB->result = MEM_GET(data_memory, memtoWB->result, ld_op, sign);
        break;
    case RISCV_ST:
        ...
        MEM_SET(data_memory, memtoWB->result, extoMem.datac, st_op);
        break;
}
```

Mem Stage

- Resolve the false inter-loop dependencies on data_memory

```
#pragma HLS DEPENDENCE variable = data_memory inter false
```

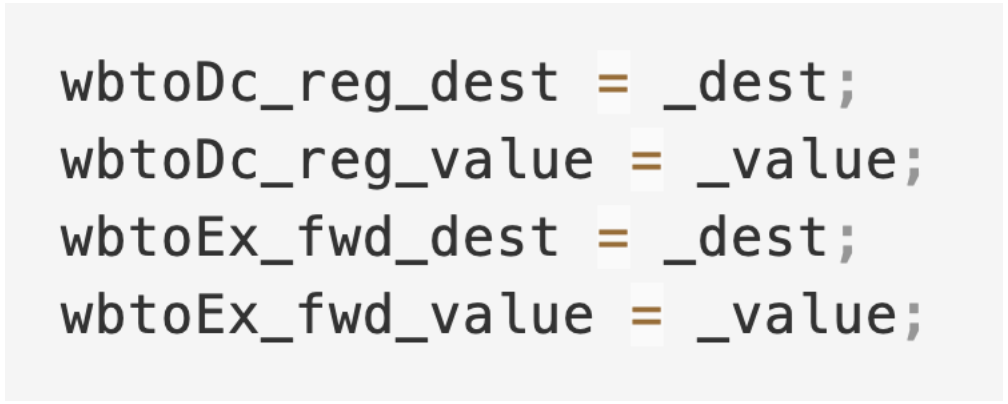
WB Stage

- WB first determine the value of _dest and _value variables, according to memtoWB
- If write-back is not needed, both variables remain 0

```
CORE_UINT(5) _dest = 0;
CORE_INT(32) _value = 0;
if (memtoWB.WBena == 1 && memtoWB.dest != 0) {
    _dest = memtoWB.dest;
    _value = memtoWB.result;
}
```

WB Stage

- The _dest and _value variables are used in Register Write-Back and Data Forwarding



```
wbtoDc_reg_dest = _dest;
wbtoDc_reg_value = _value;
wbtoEx_fwd_dest = _dest;
wbtoEx_fwd_value = _value;
```

Exiting Process

- System calls can not be handled in comet, except for the exit system call
- The exit system call should be handled, so that the processor knows when to stop the processing and exit

Exiting Process

- Similar to all other types of instruction, the exit system call instruction is classified in the switch-case in the Ex stage

```
#define EX_SYS_CALL()                   \
    case RISCV_SYSTEM:                  \
        extoMem->sys_status = 1;  \
        break;

switch (dctoEx.opCode) {
    ...
    EX_SYS_CALL()
}
```

Exiting Process

- The sys_status variable will be passed into the Mem stage, then the WB stage, so that the WB stage can set the early_exit variable to 1 when an exit instruction is processed

```
#define WB_SYS_CALL()                      \
    if (memtoWB.sys_status == 1) { \
        *early_exit = 1;                   \
    }
```

Exiting Process

- As we mentioned before, all pipeline stages are invoked in a while loop
- In fact, the while loop breaks only if
  - 1. early_exit is set to 1, or
  - 2. the cycle count reaches the max value (which is a parameter of the top function)

Exiting Process

```
while (n_inst < nbcycle && !early_exit) {
    ...
    doWB(...);
    do_Mem(...);
    Ex(...);
    DC(...);
    Ft(...);
    ...
}
```

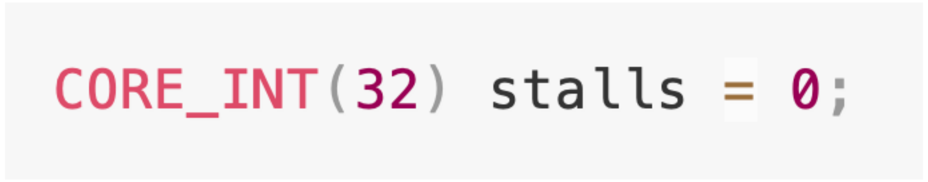
Outline

Performance Counter

- We may want to get the processing details, such as the numbers of times stall and flush occur
- Moreover, after integrating branch predictor into comet, the same instruction memory and data memory may results in different numbers of stalls and flushes due to different branch predictions
- We demonstrates the implementation of the performance counter using an example, since it is quite simple

Performance Counter

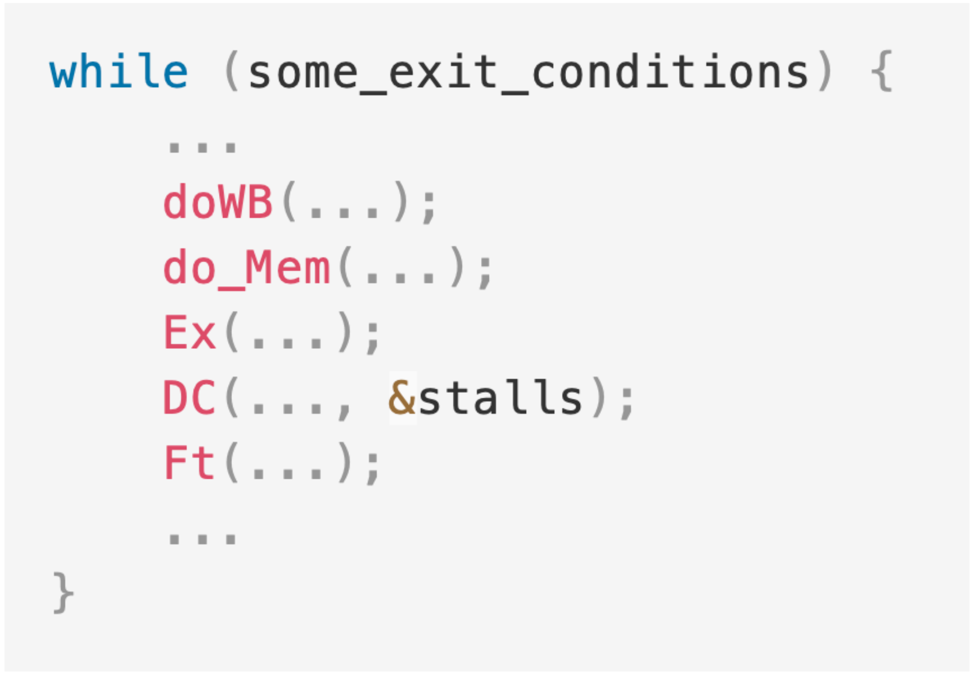
- In order to record the number of stall during processing, we use a variable, which is defined in the top function, as a counter



```
CORE_INT(32) stalls = 0;
```

Performance Counter

- Since pipeline stalls are handles in the DC stage, the counter variable is passed by pointer to the DC function



```
while (some_exit_conditions) {
    ...
    doWB(...);
    do_Mem(...);
    Ex(...);
    DC(..., &stalls);
    Ft(...);
    ...
}
```

Performance Counter

- The DC stage increases stalls by 1 when a pipeline stall occurs

```
void DC(struct FtoDC ftoDC,
        struct DCtoEx* dctoEx,
        CORE_INT(32) * stalls)
{
    ...
    if (some_stall_conditions) {
        // handling pipeline stall
        ...
        (*stalls)++;
    }
}
```

Performance Counter

- At the end of the top function, the perf_arr[] array collects all performance counters

```
perf_arr[STALL_COUNT] = stalls;
perf_arr[FLUSH_COUNT] = flushs;
...
```

Outline

- Introduction to Comet
- Implementation
    - Pipeline stages and Pipeline registers
    - Signals
    - Stall, Flush, and Data Forwarding
    - Pipeline stage implementation
    - Performance Counters
- Workflow Demonstration

Workflow Demonstration

- Compile our code into RISC-V ELF executable
- Dump out the instruction memory and data memory
- Process on FPGA

Workflow Demonstration

- Top function interfaces:
  - pc_in - s_axilite
  - pc_out - s_axilite
  - nbcycle - s_axilite
  - ins_memory_in - m_axi
  - data_memory_in_out - m_axi
  - perf_arr - m_axi

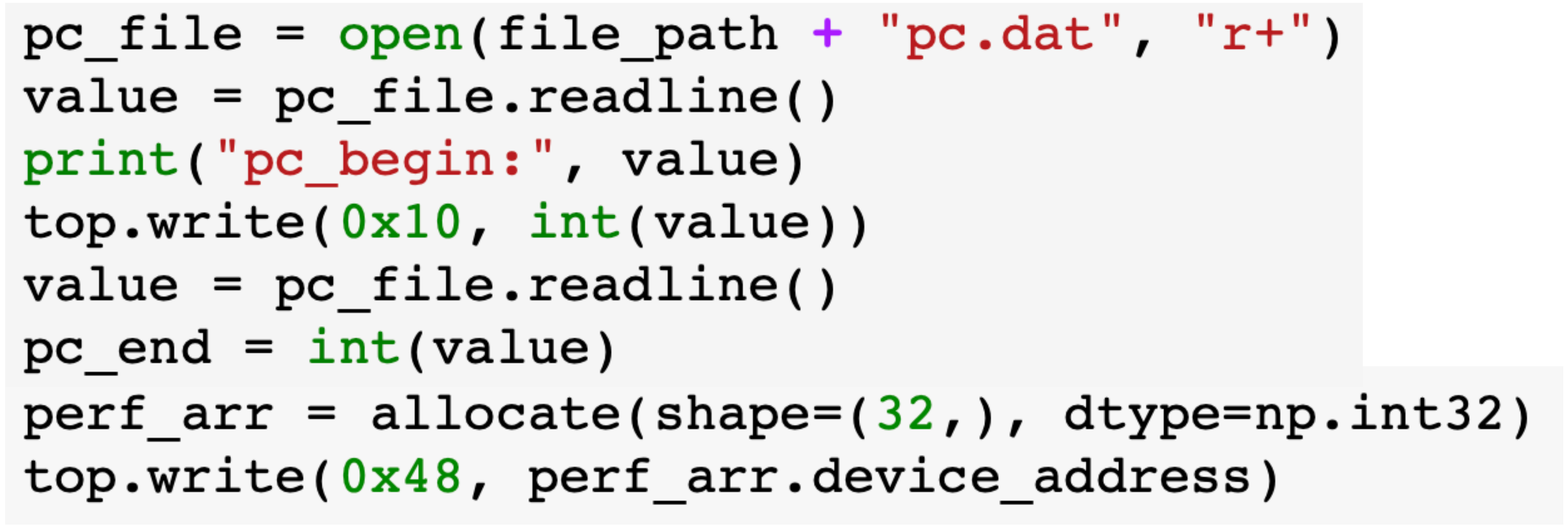
Workflow Demonstration

- Demo

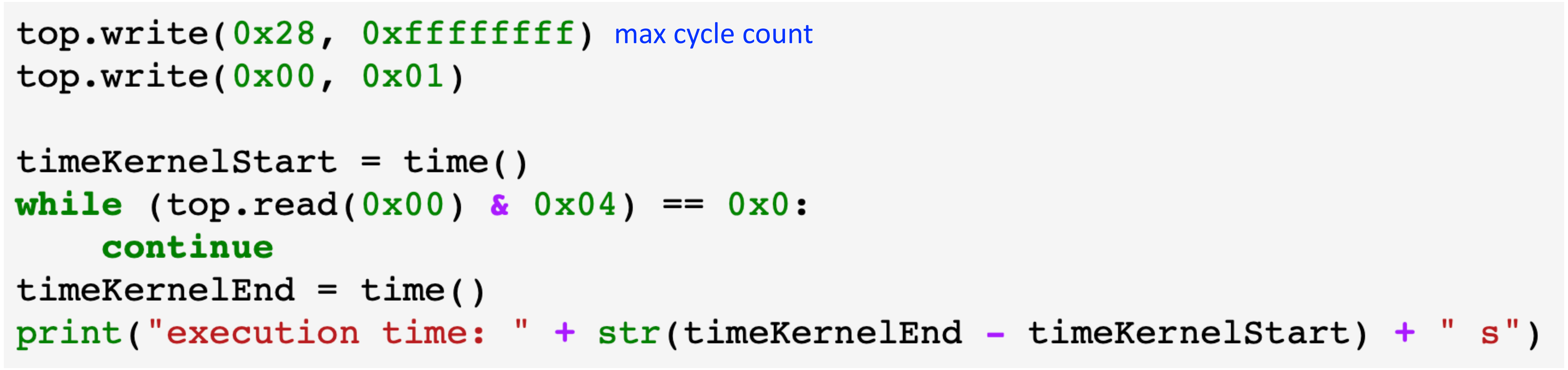
Workflow Demonstration

```
ins_memory = allocate(shape=(MEM_SIZE >> 2,), dtype=np.int32)
mem_file = open(file_path + "ins.dat", "r+")
for i in range(MEM_SIZE >> 2):
    value = mem_file.readline()
    ins_memory[i] = int(value)
top.write(0x30, ins_memory.device_address)
top.write(0x34, 0x0)

data_memory = allocate(shape=(MEM_SIZE >> 2,), dtype=np.int32)
mem_file = open(file_path + "data.dat", "r+")
for i in range(MEM_SIZE >> 2):
    value = mem_file.readline()
    data_memory[i] = int(value)
top.write(0x3c, data_memory.device_address)
top.write(0x40, 0x0)
```

Workflow Demonstration


```
pc_file = open(file_path + "pc.dat", "r+")
value = pc_file.readline()
print("pc_begin:", value)
top.write(0x10, int(value))
value = pc_file.readline()
pc_end = int(value)
perf_arr = allocate(shape=(32,), dtype=np.int32)
top.write(0x48, perf_arr.device_address)
```

Workflow Demonstration



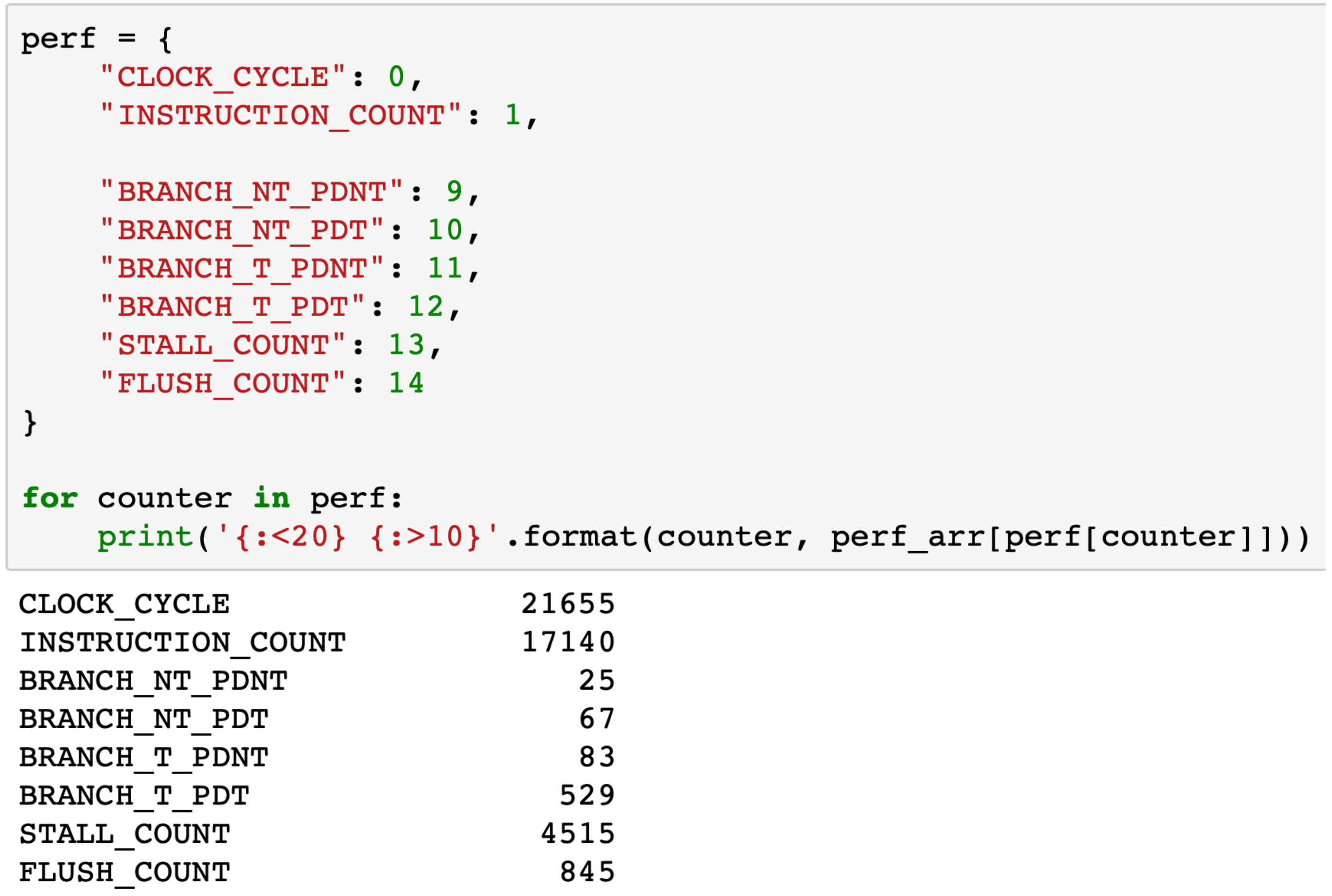
```
top.write(0x28, 0xffffffff)
top.write(0x00, 0x01)

timeKernelStart = time()
while (top.read(0x00) & 0x04) == 0x0:
    continue
timeKernelEnd = time()
print("execution time: " + str(timeKernelEnd - timeKernelStart) + " s")
```

max cycle count

Workflow Demonstration

```
mem_file = open(file_path + "data_out.dat", "w")
for i in range(MEM_SIZE >> 2):
    value = data_memory[i]
    mem_file.write(str(value) + "\n")
```

Workflow Demonstration



```
perf = {
    "CLOCK_CYCLE": 0,
    "INSTRUCTION_COUNT": 1,

    "BRANCH_NT_PDNT": 9,
    "BRANCH_NT_PDT": 10,
    "BRANCH_T_PDNT": 11,
    "BRANCH_T_PDT": 12,
    "STALL_COUNT": 13,
    "FLUSH_COUNT": 14
}

for counter in perf:
    print('{:<20} {:>10}'.format(counter, perf_arr[perf[counter]]))
```

```
CLOCK_CYCLE               21655
INSTRUCTION_COUNT         17140
BRANCH_NT_PDNT               25
BRANCH_NT_PDT                67
BRANCH_T_PDNT                83
BRANCH_T_PDT                529
STALL_COUNT                4515
FLUSH_COUNT                 845
```