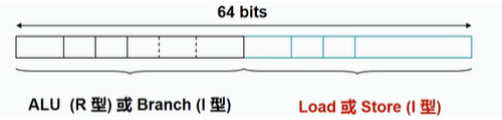
8 Instruction-Level Parallelism

8.1 VLIW

Static Multiple-Issue Processor: The compiler determines which instructions can be executed in parallel (in the same cycle).

Very Long Instruction Word (VLIW)（超长指令字）

- Combine several parallel instructions together into a very-long-instruction-word instruction.
- These instructions can executed parallelly, which means each instruction is not relevant with any other instructions.
- The **compiler** chooses and arranges the parallel instructions (use empty instruction `nop` if there is no feasible instruction).

> [*Example*] An instruction includes two basic instructions.
>
> 
>
>
> Here is an assembly program.
>
> ```
> lp:  lw   $t0,0($s1)  # $t0=array element
>      addu $t0,$t0,$s2  # add scalar in $s2
>      sw   $t0,0($s1)   # store result
>      addi $s1,$s1,-4   # decrement pointer
>      bne  $s1,$0,lp    # branch if $s1 != 0
> ```
>
> And here are the new "instructions".
>
	ALU 或 branch	数据传送	时钟周期
lp:		lw $t0,0($s1)	1
	addi $s1,$s1,-4		2
	addu $t0,$t0,$s2		3
	bne $s1,$0,lp	sw $t0,4($s1)	4
>
> Suppose the branch predictions are always correct, then $CPI = 0.8$

Loop Unrolling（循环展开）: sometimes we can use loop unrolling to have more instructions that can be executed parallelly.

- Improve the ILP (instruction level parallelism).
- Less jump/branch instructions, lower cost;
- **Rename registers** to solve some 'fake' relevant problems (because of using the same register in loop process, actually we can use different registers).

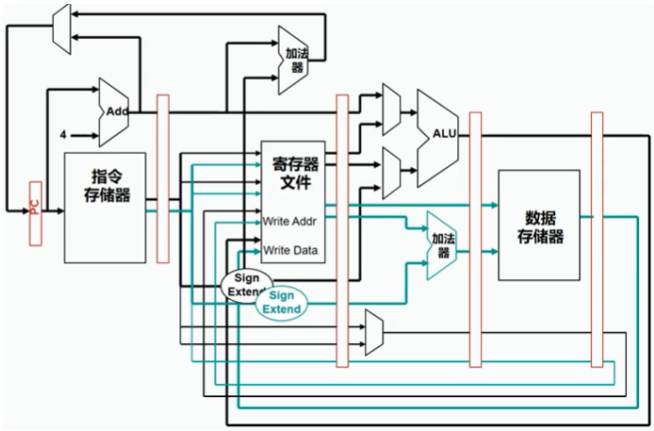
> [*Example*] We first use loop unrolling to unroll the loop.
>
> ```
> lp:
>   lw      $t0,0($s1)
>   addu    $t0,$t0,$s2
>   sw      $t0,0($s1)
>   addi    $s1,$s1,-4
>   bne     $s1,$0,lp
> ```
>
> 循环展开 ⇨
>
> ```
> lp:  lw   $t0,0($s1)
>      addu $t0,$t0,$s2
>      sw   $t0,0($s1)
>      lw   $t1,-4($s1)
>      addu $t1,$t1,$s2
>      sw   $t1,-4($s1)
>      lw   $t2,-8($s1)
>      addu $t2,$t2,$s2
>      sw   $t2,-8($s1)
>      lw   $t3,-12($s1)
>      addu $t3,$t3,$s2
>      sw   $t3,-12($s1)
>      addi $s1,$s1,-16
>      bne  $s1,$0,lp
> ```

> And then we can change the order to let more instructions execute in parallel.

```
lp:  lw   $t0,0($s1)
     lw   $t1,-4($s1)
     lw   $t2,-8($s1)
     lw   $t3,-12($s1)
     addu $t0,$t0,$s2
     addu $t1,$t1,$s2
     addu $t2,$t2,$s2
     addu $t3,$t3,$s2
     sw   $t0,0($s1)
     sw   $t1,-4($s1)
     sw   $t2,-8($s1)
     sw   $t3,-12($s1)
     addi $s1,$s1,-16
     bne  $s1,$0,lp
```

ALU or branch	Data transfer	CC
addi $s1,$s1,-16	lw $t0,0($s1)	1
	lw $t1,12($s1)	2
addu $t0,$t0,$s2	lw $t2,8($s1)	3
addu $t1,$t1,$s2	lw $t3,4($s1)	4
addu $t2,$t2,$s2	sw $t0,16($s1)	5
addu $t3,$t3,$s2	sw $t1,12($s1)	6
	sw $t2,8($s1)	7
bne $s1,$0,lp	sw $t3,4($s1)	8

VLIW circuit structure: simple, only need to add some similar lines and devices.



Disadvantage

- **Complex Complier**
  - Loop unrolling, conflict detection, instruction dispatch;
  - Branch prediction;
  - Memory access address prediction;
  - Result in *longer compile time*.
- **Code Extension**（代码膨胀）
  - `NOP` instructions waste spaces;
  - Loop unrolling wastes spaces;
  - Need wider instruction path.
- **Lock step**（锁步机制）
  - One instruction is blocked, then all the instructions after it is blocked;
  - The new instructions are allowed to issue only when there is no relevant situation anymore.
- **Low-Compatibility**（兼容性差）
  - Need a special compiler and more complex instructions;
  - Market is unwilling to accept it.

Advantage

- Simple hardware;
- Low power-consume;
- Low cost.

Usage

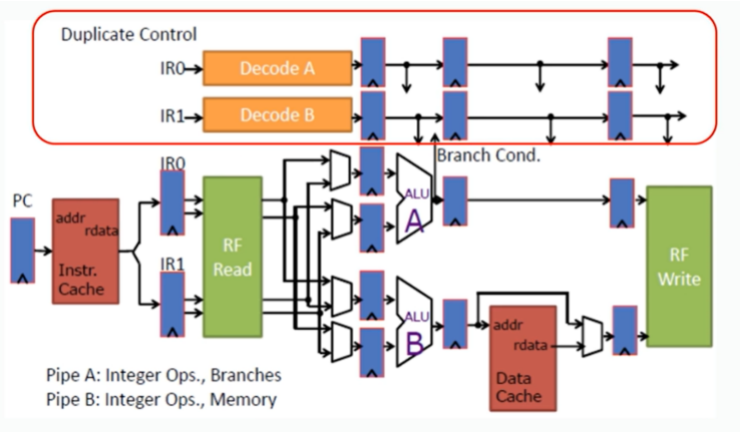
- some special fields, such as signal processor.
- **EPIC technique**（显式并发指令运算）: Intel Itanium Processor (2002).

- Bad performance comparing to superscalar processor.
- Rarely used now.

8.2 In-order Superscalar Processor

Dynamic Multiple-Issue Processor (a.k.a., superscalar processor，超标量处理器): use hardware to detect and solve conflict problems.

In-order Superscalar: execute the instructions in order.



There is a duplicate control module used to detect the relations between instructions, and if the current instruction is relevant with another one, then we will block the current instruction.

- Can **swap the order of the parallel instructions** to fit the structure of processor (e.g., in the picture above, put the load/store instruction in part B);
- Can **detect the structural hazard and data hazard**;
- Can add forward path (bypassing).

Example: ARM A8 processor.

Features and Applications

- Less pipeline stage, low frequency.
- Mobile, laptop, embedded system.

8.3 Out-of-order Superscalar Processor

Out-of-order:

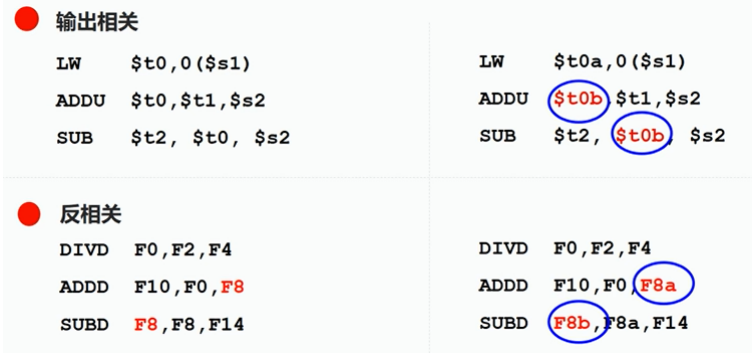
- The fast instruction doesn't have to wait the slow instruction;
- If the current instruction have relevant with other instructions, then we can execute other irrelevant instructions first (the instructions after the current one).

Fake data relevance

- Output relevance（输出相关） **Write-After-Write, WAW**

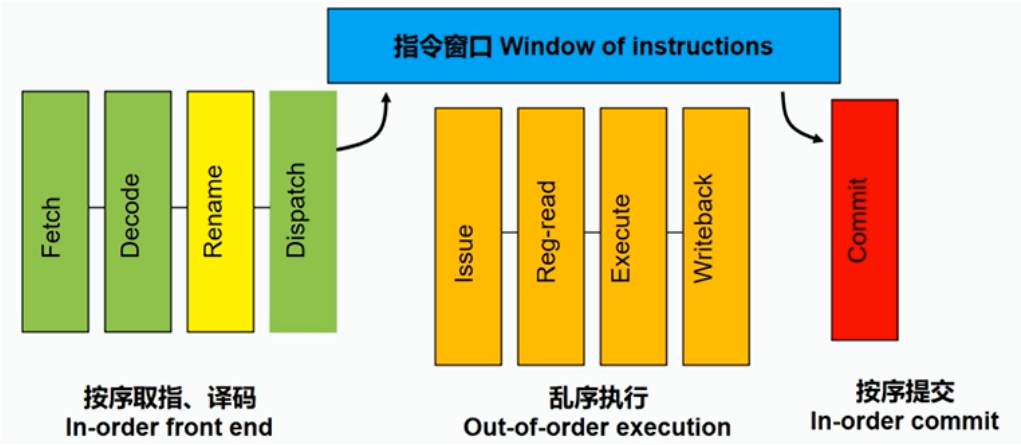
- Reversed relevance（反相关）**Write-After-Read, WAR**

Solve fake data relevance: Re-name registers.



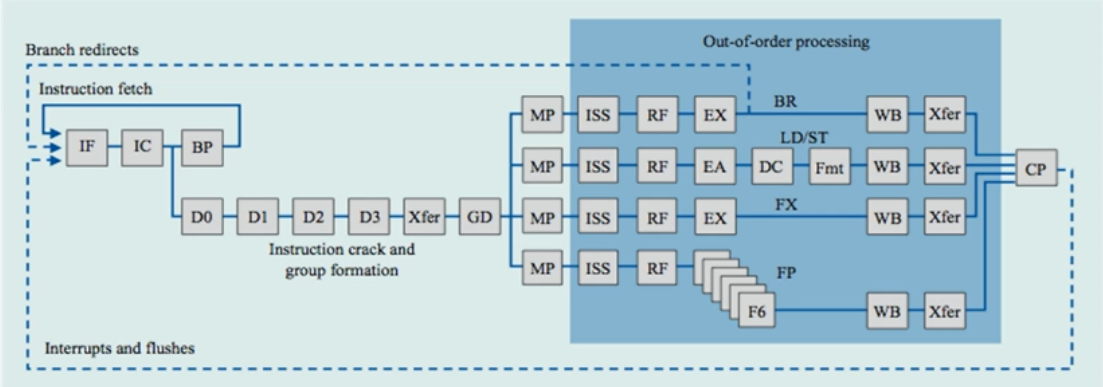
Real data relevance（真相关） **Read-After-Write, RAW**: still need to pause the current instruction or add forwarding path.

Out-of-order pipeline



- In-order front end, Out-of-order execution, In-order commit;
- Write back and memory access stages only operate on the data in the **buffer**（缓冲区）(data-read stage also needs to consider the data in the buffer);
- The modifications of data are **committed** in the last step **in order**.
- Handle exceptions **before commit** (not before write back).

Example: IBM Power4.



Instruction pipeline (IF: instruction fetch, IC: instruction cache, BP: branch predict, D0: decode stage 0, Xfer: transfer, GD: group dispatch, MP: mapping, ISS: instruction issue, RF: register file read, EX: execute, EA: compute address, DC: data caches, F6: six-cycle floating-point execution pipe, Fmt: data format, WB: write back, and CP: group commit)

8.4 Branch Prediction

Control Hazards

- MIPS (5-stage): unconditional branch instruction $CPI_{stall} = 3$.

- The longer the pipeline is, the more expensive the cost of control hazard is.
  - Pentium 3: $CPI_{stall} = 10$.
- For multi-issue processor, the cost of branch equals to $\text{cost} \times \text{issue width}$.

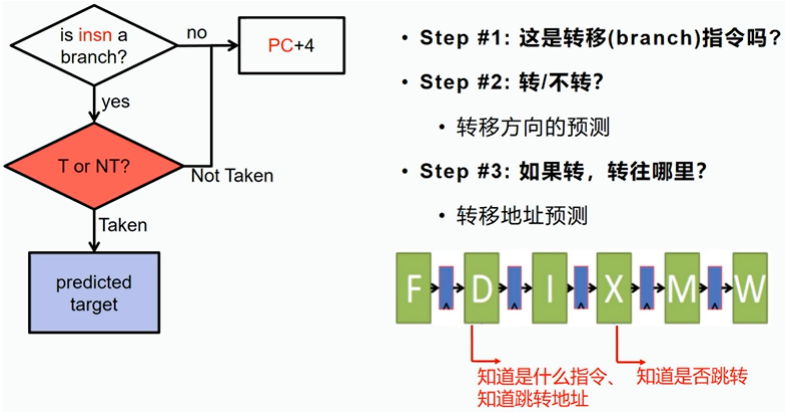
Current Solution

- Pause: the cost is too expensive.
- Branch slot（转移延迟槽）: it is depends on the compiler, and the compiler sometimes can not find enough instructions to fill the slot.
- **Branch Prediction, BP**

Branch Prediction Categories

- Static prediction (by compiler): predict before the execution of program; (not-so-good performance)
- Dynamic prediction: predict after the execution of program. We mainly discuss this one.

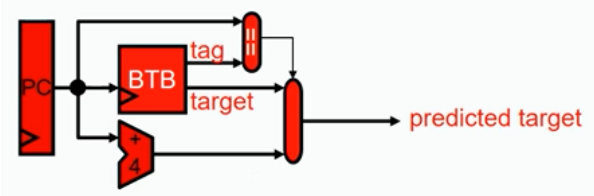
Dynamic prediction



Branch Target Buffer, BTB（转移目标历史表）

- Use the historical data to predict the future address.
- Last time from branch X to address Y, then this time we assume the next instruction is still in address Y.
- Use the last few bits of `PC` as tag to search BTB.
  - Multiple `PC` with the same tag? Don't worry, just prediction.
- For every branch instruction, update the record in BTB.

```
BTB[PC].tag = PC;
BTB[PC].target = target of branch;
```



- In Instruction Fetch stage will read BTB parallelly, and check for tag. If the tag equals to the current `PC`, then we can predict the address after jump.

```
Predicted PC = (BTB[PC].tag == PC) ? BTB[PC].target : PC + 4;
```

BTB is effective for direct transfer, but what about **indirect transfer**?

- `JR`: the address is in register;
- Function call:
  - Dynamic Linking (DLLs，动态链接): the target is the same every time.
  - Virtual function call（虚拟函数调用）: hard to predict, but rarely to be seen.
- Function return: hard to predict, but we can use **RAS**!

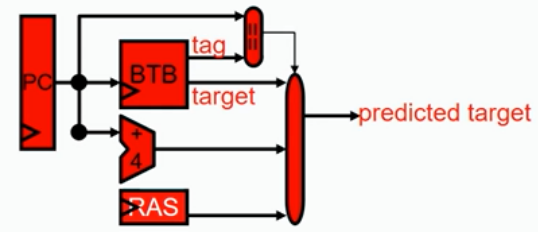
Return Address Stack, RAS（转移地址栈）

- Function Call:

```
RAS[Top++] = PC+4;
```

- Function Return:

```
Predicted PC = RAS[--Top]
```



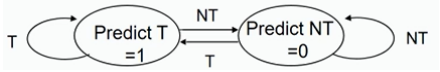
- How can we know the type of instruction (function call, function return) before decoding?

  Possible Solutions

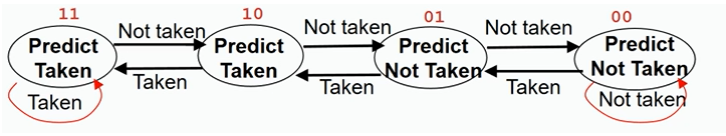
  - Use another **predictor** to memorize the function call and function return address;
  - Add **"return" tag** in BTB;
  - Add **pre-decode bits**（预译码位） in the instruction memory.

Branch History Table, BHT（转移历史表）

- Use the historical data to predict the branch outcome (jump or not).
- Similar with BTB, in Instruction Fetch stage will read BTB parallelly, and check for tag (`T` or `NT`). If tag is `T` then we predict to jump.
- Use the last few bits of `PC` as tag to search BHT.
  - Multiple `PC` with the same tag? Don't worry, just prediction.
- **One-bit Predictor**: `0` or `NT` means not jump, and `1` or `T` means jump. Only depend on the last branch result of this instruction.



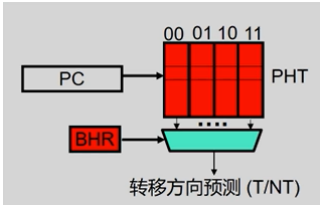
- **Two-bit Predictor**: `00` means strong `NT`, `01` means weak `NT`, `10` means weak `T`, and `11` means weak `NT`. We can use a `count` variable to implement this process, if taken then `count = min(count + 1, 3)` and if not taken, `count = max(count - 1, 0)`.



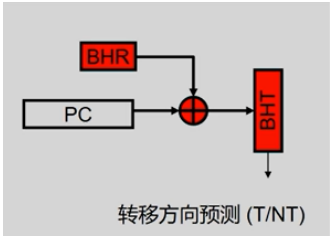
Sometime one-bit predictor and two-bit predictor have poor performance because the branch instruction can be locally relevant（局部相关）, so we can use the historical pattern to predict. Therefore, we can use a **BHR** to solve the problem.

Branch History Register, BHR（转移历史寄存器）

- Use the historical *pattern* to predict the branch outcome (jump or not).
- Record the branch outcomes pattern.
- **Pattern History Table, PHT** (BHT for different patterns).

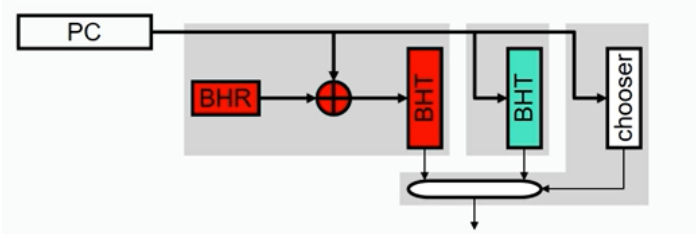


  - Use the patterns in BHR to find the corresponding predictor in PHT.
  - Use the predictor to predictor the result.
  - High power-consume and it needs large space.
- **G-share Branch Prediction**: use `PC xor BHR` to search BHT, low power-consume, often used in embedded systems.



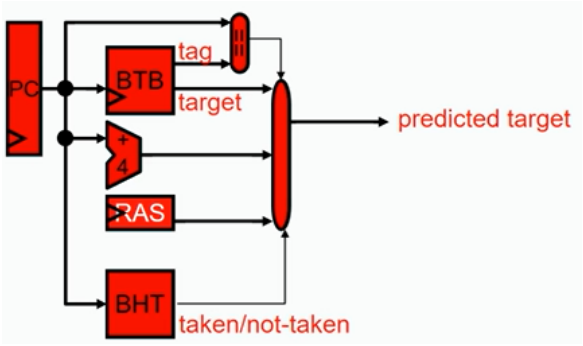
> [*Example*] G-share branch prediction.
>
> ```
> for (i=0; i<1000000; i++) {
>   if (i % 4 == 0) {
>     ...
>   } ...
> }
> ```
>
> 假设：
> - 指令PC：最后10位都是0
> - BHT： 一位预测器, 初始状态为NT
>
> | Branch History | Prediction |
> |---|---|
> | TTT | N |
> | TTN | T |
> | TNT | T |
> | NTT | T |
>
> | Time | State | BHR | Prediction | Outcome | Result? |
> |---|---|---|---|---|---|
> | 0 | N | NNN | N | T | wrong |
> | 1 | N | NNT | N | T | wrong |
> | 2 | N | NTT | N | T | wrong |
> | 3 | N | TTT | N | N | correct |
> | 4 | N | TTN | N | T | wrong |
> | 5 | N | TNT | N | T | wrong |
> | 6 | T | NTT | T | T | correct |
> | 7 | N | TTT | N | N | correct |
> | 8 | T | TTN | T | T | correct |
> | 9 | T | TNT | T | T | correct |
> | 10 | T | NTT | T | T | correct |
> | 11 | N | TTT | N | N | correct |

Championship Predictor（锦标赛预测器）: combine two predictors together.



- Simple predictor（简单预测器）: 1-bit or 2-bit, for history-independent branches;
- History correlated predictor（历史相关预测器）, for branches that need history.
- Chooser（选择器）:
  - Choose simple BHT at first, and after some `threshold`, choose the history correlated predictor.
  - Similar to the ensemble learning（集成学习） in machine learning.
- 90% ~ 95% accuracy.

Final Design: combine BTB and BHT together.

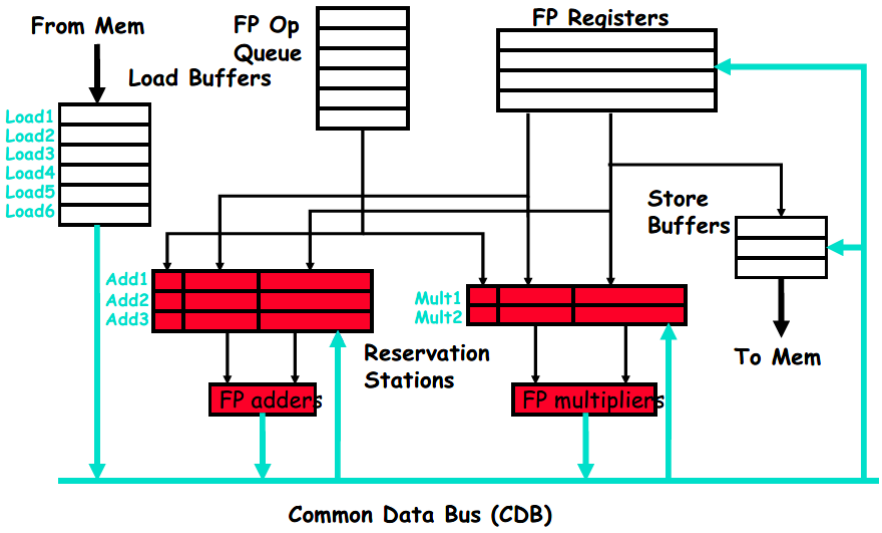


- If the prediction is right, there is no extra cost in branch instructions;
- If the prediction is wrong, then the wrong instruction cannot affect the state because it cannot be committed because of in-order commitment. (Once we find our prediction is wrong, we immediately flush the wrong instructions.)

8.5 A Dynamic Algorithm: Tomasulo Algorithm

Goal *High performance* without special compilers (mainly by hardware).

Tomasulo Organization



- Control & buffers **distributed** with Function Units (FU); FU buffers are called **reservation station**（保留站, RS） storing pending operands.
- Registers in instruction replaced by *values of pointers to RS*, which is called **register renaming**; It can avoid WAR and WAW hazards.
- Results to FU from RS, **not through registers**, over **Common Data Bus** that broadcasts（广播） result to all FUs (similar to forwarding path).

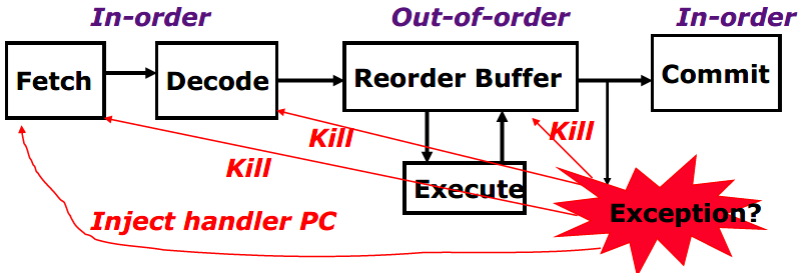
Three Steps of Tomasulo Algorithm

- **Issue**（发射）: get instruction from FP Op Queue. If reservation station free (*no structural hazard*), control issues instruction & send operands (renames registers, if some *operands* are ready, then directly store them in RS).
- **Execution**（执行）: operate on operands (EX). When both operands ready then execute; if not ready, watch Common Data Bus for result.
- **Write result**（写结果）: finish execution (WB). Write on Common Data Bus to all awaiting units, and mark reservation station available.

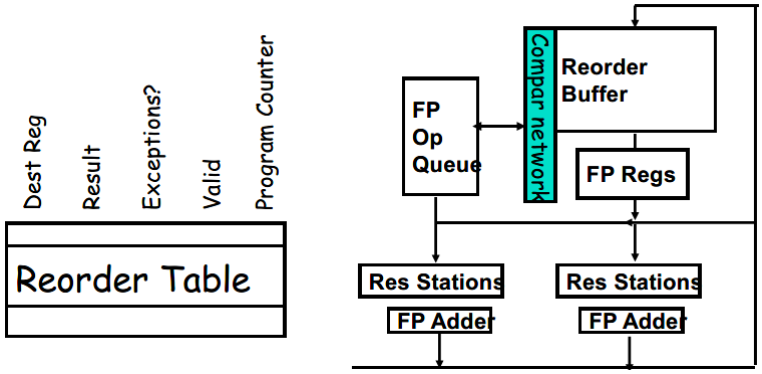
Tomasulo Example: refer to `Tomasulo-Examples/Tomasulo-Basic-Examples.pdf` for details.

Tomasulo Loop Example: refer to `Tomasulo-Examples/Tomasulo-Loop-Examples.pdf` for details. Actually, Tomasulo Algorithms can *unroll some loops automatically* (out-of-order executing).

Reorder Buffer（重排序缓冲区）: Add a reorder table before commitment to solve the precise interrupts problems, and commit the result in order.



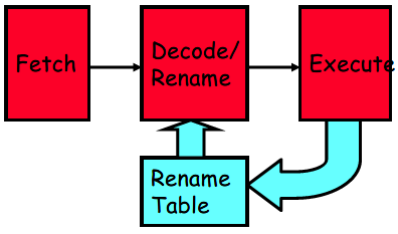
Therefore, we need to fetch some operands from Reorder Buffer, and our new organization is as follows.



Four Steps of (Improved) Tomasulo Algorithm

- **Issue**（发射）: get instruction from FP Op Queue. If reservation station *and reorder buffer slot* free（重排序缓冲区有空间）(*no structural hazard*), control issues instruction & send operands (renames registers, if some *operands* are ready, then directly store them in RS) *& reorder buffer no. for destination*（在重排序缓冲区中开辟空间存储这条指令的结果）.
- **Execution**（执行）: operate on operands (EX). When both operands ready then execute; if not ready, watch Common Data Bus for result.
- **Write result**（写结果）: finish execution (WB). Write on Common Data Bus to all awaiting units *& reorder buffer*, and mark reservation station available.
- **Commit**（提交）: update register with *reorder result*. When instruction at head of reorder buffer & result present, update register with result (or store to memory) and remove instruction from reorder buffer. *Mis-predicted branch flushes reorder buffer (called graduation)*（有异常或错误预测的指令将会清除重排序缓冲区的错误指令以下的所有内容）.

Explicit Renaming



- Free registers in *register free list*, can be used for renaming.
- Examples: refer to `Tomasulo-Examples/Explicit-Renaming-Examples.pdf` for details.
- Advantages:
    - Decouple renaming from scheduling;

- Allows data to be fetched from single register file;
- Many processors use a variant of this technique;
- Another way to get precise interrupt point.

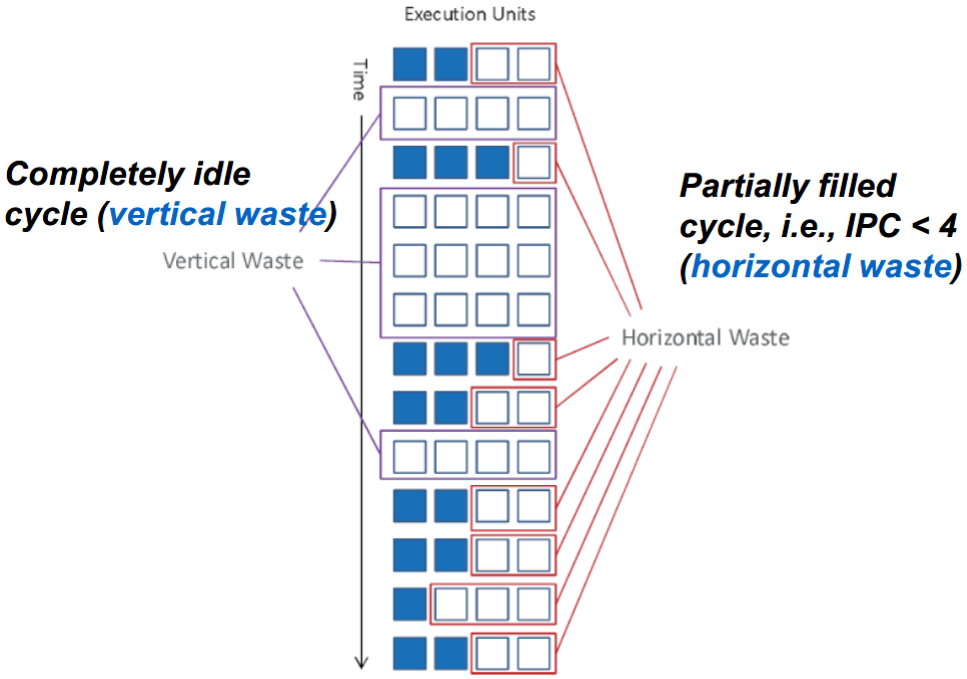
No store buffer, only load buffer: store buffer is just like reorder buffer.

8.6 From ILP to TLP

TLP (Thread Level Parallelism, 线程级并行).

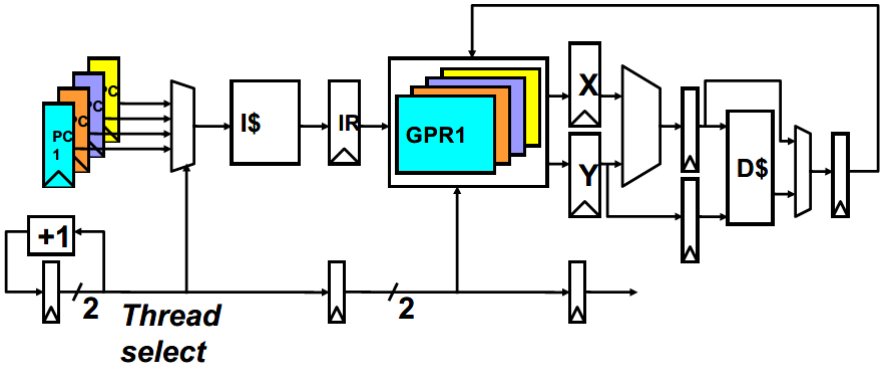
Restrictions of ILP

- Programmers write program in sequential order, relevance happens frequently.
- Many operations need to be scheduled, therefore the latency may increase.
- Hardware cost:
  - May need more ports of register files (large bandwidth);
  - May need more ports of memory files (large bandwidth).
- **Superscalar Waste**



  - Completely idle circle (vertical waste): because of interruptions, relevance and so on;
  - Partially filled cycle (horizontal waste): because of relevance.

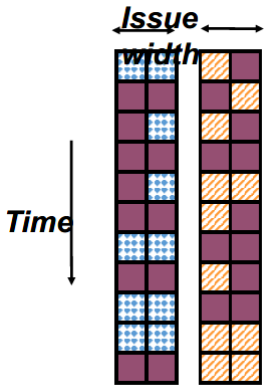
Multithreading



- Execute the instructions from different threads in the same pipeline.
  - The relevance problem is solved.

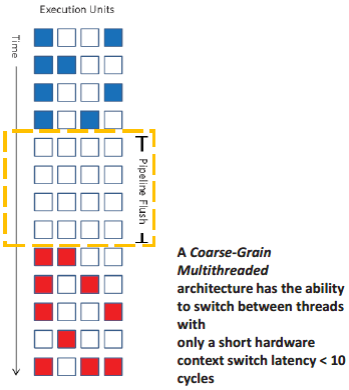
- Must have thread-select signal to select the thread to execute.
- From OS view, it seems like there are many CPUs.
- For every thread, we need to store
    - The user state (PC, GPRs);
    - The system state (Virtual-memory page-table-base register, Exception handling register);
- Need to handle the conflict of threads in cache / TLB (increase the capacity of TLB & cache);
- More OS scheduling cost.

Chip Multiprocessing (CMP)

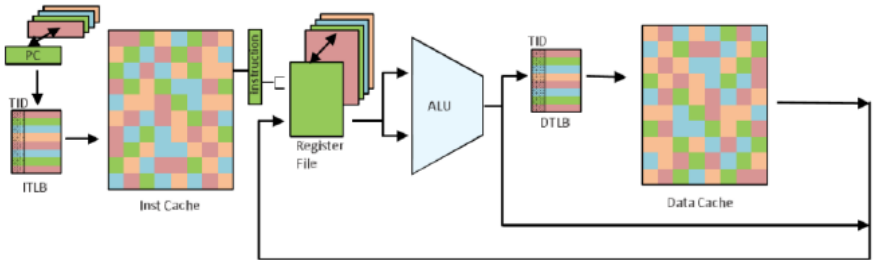


- From every core's view, the horizontal waste & vertical waste is not reduced;
- From the whole view, the horizontal waste & vertical waste is, to some extent, reduced.

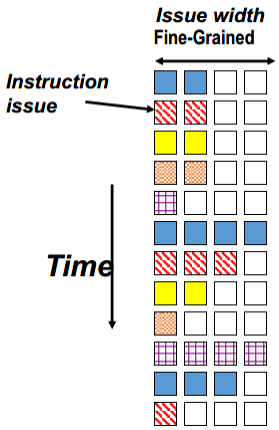
Vertical (Coarse-Grain) Multithreading (CGTM, 粗粒度多线程方式)



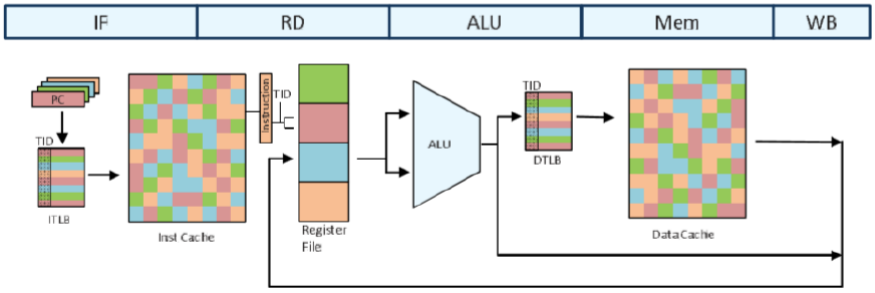
- Only when a thread has a long latency in execution（线程需要延时较长时间），perform context switch to another thread (need to save the state of last thread).
- Still have vertical waste because of context switch.
    - **Solution**: design a register file for each thread, therefore we do not need to store the state because the state is already in register (or design a mapping to map the thread to its registers, like the register in SMT's architecture).
- **Architecture**:



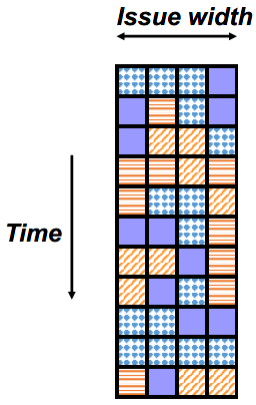
Fine-Grain Multithreading (FGTM, 细粒度多线程方式)



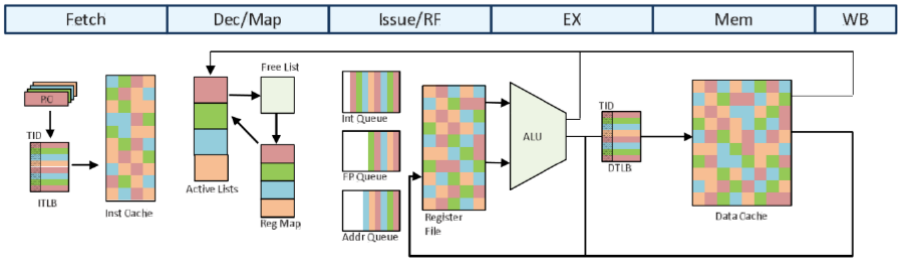
- Perform context switch after a fixed period of time（定时切换）.
- Reduce & eliminate vertical waste.
- Still have horizontal waste.
- **Architecture**: (there is an implicit mapping from thread to registers in register file, because the thread is execute in certain order, so we only need to arrange the registers in the same order)



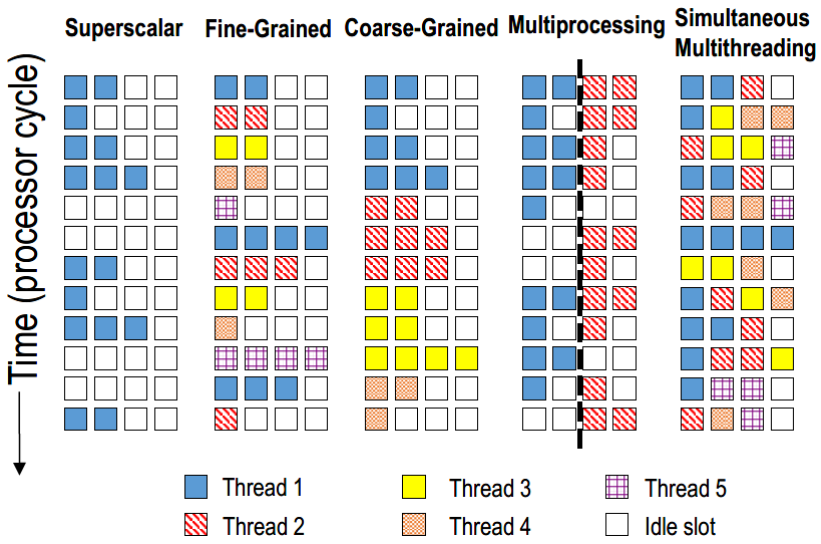
Out-of-Order Simultaneous Multithreading (OoO SMT)



- Issue instructions from multiple threads at the same time to eliminate both horizontal waste & vertical waste.
- This architecture can fully use the ILP and TLP.
- Need **register renaming** for every thread to access register parallelly (eliminate the register name conflict for different thread).
- **Architecture**: (there is an explicit mapping from thread to registers in register file)


Fetch
Dec/Map
Issue/RF
EX
Mem
WB
PC
TID
ITLB
Inst Cache
Free List
Active Lists
Reg Map
Int Queue
FP Queue
Addr Queue
Register File
ALU
TID
DTLB
Data Cache

Summary


Superscalar
Fine-Grained
Coarse-Grained
Multiprocessing
Simultaneous Multithreading
Time (processor cycle)
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
Idle slot