Presentation 3

Directions for Improving Performance

מעבדים בסגנון RISC הצליחו לשפר ביצועים בצורה משמעותית ביחס למעבדים ישנים יותר. אם בשנת 1985 מעבד CISC פעל בקצב שעון של 20 MHz עם CPI בסביבות 8, בשנת 2000 מעבד RISC פעל בקצב שעון של 4 GHz עם CPI בסביבות 1. ביחד זה שיפור של פי 1600 בזמן ריצה. כדי להמשיך לשפר את הביצועים חייבים להשיג מידה משמעותית של מקבילות, גם ברמת הפקודות וגם ברמת הפתיל (thread). הפרק הזה עוסק בשיטות להשיג מקבילות רבה ברמת הפקודות.

Slide 2 **General Outlook**

The **fundamental parameters** of performance follow from the equation $T = CPI \times IC \times \tau$. Reducing any of the three factors improves run time. But this desire confronts certain technological realities:

1. IC grows as software becomes more complex. Not only are programs getting larger, but software development favors fast development over efficient run time. The use of advanced programming techniques, such as middleware, OOP, design patterns, and virtualization, introduce additional layers of code translation that contribute to IC at run time.
2. The clock period τ is reaching physical limits. The time required for an electrical signal to cross a 3 cm wide chip is 3 cm / (3×10^{10} cm/sec) = 10^{-10} seconds, where 3×10^{10} cm/sec is the speed of light. This is the period of a 10 GHz clock, since $(10\ GHz)^{-1} = 10^{-10}$ seconds.
3. For a RISC processor, $CPI^{ideal} = 1$. A sequential processor cannot improve on this.
4. CPI^{stall} is affected by data dependencies, memory latency such as cache miss, and control hazards in branches.

The areas for possible improvement are:

1. Achieving $CPI^{ideal} < 1$ with instruction and thread level parallelism
2. Reducing instruction dependency stalls to lower $CPI^{stall}_{data\ dependency}$
3. Reducing cache latency to lower $CPI^{stall}_{cache\ miss}$
4. Reducing branch stalls to lower $CPI^{stall}_{branch\ penalty}$

This chapter discusses **Instruction Level Parallelism** (ILP) and reduction of $CPI^{stall}_{data\ dependency}$.

Slide 3 **Processing Parallelism**

Processing can be made parallel in two ways:

1. **Instruction Level Parallelism** (ILP) — parallel execution of instructions in a single thread
2. **Thread Level Parallelism** (TLP) — parallel execution of a single program by multithreading

ILP includes

1. **Pipelining** — multiple sequential instructions from a single thread in various stages of execution at any given time (as in the DLX)

2. **Superscalar** processing — multiple sequential instructions from a single thread executing in parallel on multiple execution units of a single CPU

Multithreading does not imply parallelism. **Concurrency** describes multiple threads that are defined in software to run independently. Concurrent threads can run in **parallel** (on separate cores) or by **interleaving** (threads executing by taking alternate turns on the same CPU core).

TLP refers to **multiprocessing** — multiple independent processor cores running independent threads.

Slide 4 – 7 Flynn Taxonomy

The **Flynn Taxonomy** enumerates the 4 possible ways of combining data and instruction streams. Each stream may come from a single source or multiple sources:

1. Single Instruction Single Data (SISD) — a single operation on a single pair of operands
2. Single Instruction Multiple Data (SIMD) — single operation on multiple operands, such as a vector processor (slide 8)
3. Multiple Instruction Single Data (MISD) — multiple operations on single operand, such as the IBM Watson IA that applies multiple algorithms to the same data
4. Multiple Instruction Multiple Data (MIMD) — multiple operations on multiple operands, such as implemented in a multiprocessor, multithreaded processor, and cluster computer.

The DLX discussed in presentation 2 is an example of SISD. This chapter focuses on ILP, which includes SIMD and superscalar methods. Approaches to MIMD will be discussed in presentations 7 – 9.

Slide 8 – 11 SIMD — Vector ALU

A **vector processor** receives multiple pairs of operands per clock cycle and performs the same operation on each pair. In the example on slide 8, a pair of vector registers (each 4 operands wide) receives 4 pairs of operands, and performs 4 addition operations in parallel.

Slide 9 shows the DLX updated to include a vector ALU and parallel registers P1, P2, ... , P31. The code loads 4 operands in parallel, loads another 4 operands in parallel, and then adds 4 pairs of operands. The execution follows the regular pattern for DLX (including the Load-ALU stall).

Some practical examples of a vector ALU include the Intel MMX, SSE, and AVX ALUs, and the AltiVec ALU in the IBM Power ISA. Compilers for these ISAs support vectorization for appropriate data structures.

The sample program on slide 11 shows vectorization of a routine that adds the 100 elements of an array. The standard SISD implementation performs 100 loop iterations. The SIMD vector implementation operates on 4 consecutive elements in each of 25 iterations — this produces 4 partial sums. The last step is called horizontal add — it produces the total from the 4 partial sums.

Slide 12 – 13 Optimizing Data Structures

A compiler requires appropriate **data structures** to support the vector processor.

Consider a 3D graphic object described by 200 vertices and 200 color pixels. A programmer might be tempted to describe this object in an array of structures (AoS):

```
struct { float x, y, z, r, g, b } AoS_xyz_rgb[200];
```

with 200 elements that describe one 3D vertex (x,y,z) and one pixel (r,g,b). But the vector processor cannot access successive operands for parallel execution. The first 4 operands are **`x[0], y[0], z[0], r[0]`**, which cannot be meaningfully processed with the next 4 operands, **`g[0], b[0], x[1], y[1]`**.

The **hybrid data structure** on slide 13 first separates the vertices and the pixels into separate structures — these can be processed by separate threads. Each structure is an array of 50 structs — each struct holds 4 consecutive values of one coordinate or color. The vector processor can then operate on 4 **x** values at one time: **`x[0], x[1], x[2], x[3]`**, followed by 4 **y** values, and so on.

Slide 14 – 16 Graphics Processing Unit

The **Graphics Processing Unit** (GPU) is a specialized graphics processor usually located separately from the CPU (on the computer's main board or a separate graphics card). The CPU can offload graphics operations to the GPU as an auxiliary processor.

The GPU is a complex system beyond the scope of this course (and requires a background in graphics processing to understand its fundamental terms). This section is intended only to introduce the subject in a basic way.

A typical x86 CPU and a typical Nvidia GPU are about the same size, approximately 3 billion transistors. The main difference is that the CPU is optimized for the efficient processing of sequential code (including different types of ALU and significant resources for cache memory), while the GPU consists of a very large number of identical execution cores.

GPUs have undergone a three-step evolution.

In **Step 1**, the execution core was a **hardwired graphics pipeline**, designed to perform the standard operations required to render a 3D object described by vertices into a graphic image. The stages were:

1. Geometry —transformation of a set of 3D **vertices** (x, y, z) into a description by 2D **triangles**, the area enclosed by sets of three vertices.
2. Rasterization —transformation of the **triangles** into **fragments**, a description in terms of computer screen coordinates (**raster**) of a set of points with a single color.
3. Fragment — color interpolation between neighboring fragments to produce a texture made of colored fragments.
4. Composition — transformation of the textures into an image, a set of pixel coordinates and colors appropriate for display on the screen

In this model, the application provides the GPU with 3D data, and the GPU renders the data into a graphics image for display.

In **Step 2**, the hardwired graphics pipeline was replaced by a **programmable pipeline**, in which the application provides the GPU with a command to perform on a set of vertices. The command allows application control over the standard pipeline stages found in the hardwired pipeline.

In **Step 3**, the programmable pipeline was replaced by a **shader**, a completely programmable processor core with hardware optimized to support graphics operations.

On the simple repetitive operations appropriate to a GPU, its performance is about 4 times faster than a standard single core CPU of its era.

The maximum **theoretical performance** of the vector processor on a single core CPU (the SSE on a 3.7 GHz Pentium 4) can be found as:

1 pipeline $\times$ 4-wide SIMD $\times$ 3.7 GHz = 14.8 GFLOPS (Giga Floating Point Operations Per Second).

The maximum **theoretical performance** of the Nvidia GeForce 6800 Ultra can be found as:

16 pipelines $\times$ 8-wide SIMD $\times$ 520 MHz = 63 GFLOPS

Slide 16 shows the structure of an Nvidia General Purpose GPU (GPGPU). The **shader** element receives:

1. Graphics data from graphics cache that accesses general memory,
2. Instructions from the application — instructions are processed using a library of graphics routines (implemented in silicon).

The Nvidia GeForce 8800 GPU (2005) consists of 8 thread processors with 16 shaders/processor, for a total of 128 shaders.

The Nvidia Kepler GPU (2012) consists of 15 SMX streaming multiprocessors with 192 shaders per SMX, for a total of 2880 shaders.

Slide 17 Superscalar Pipeline Models for DLX

A superscalar provides explicit ILP by initiating multiple instructions at the same time (in the same clock cycle). There are two possible approaches:

1. **Replicate the entire pipeline** and launch a pair of instructions on each CC — one instruction in each pipeline. Each pipeline operates in the same way as on a single pipeline CPU. This approach is difficult to implement and was only attempted on one mass-market commercial processor — the Intel Pentium in 1993.
2. **Replicate the execution units** (EUs) and send instructions to EUs as needed. In this model, the IF and ID stages operate quickly to fetch and decode multiple instructions per clock cycle. The decoded instructions are stored in an **instruction pool**, a large register holding the instructions along with their source operands (read from the registers). A controller sends the pooled instructions to EUs as needed.

Slide 18 – 20 Superscaling in Intel Pentium

The Pentium included two pipelines of the type designed for the Intel 486 CPU. The 486 pipeline stages are somewhat different from a RISC pipeline:

1. Instruction Fetch (IF)
2. Stage-1 Decode (D1) — identify the operation, read register sources and calculate the register part of the address for memory sources
3. Stage-2 Decode (D2) — calculate the physical data address and generate the microcode required to execute the instruction
4. Execution (EX) — integer ALU operations, FPU operations, and data memory accesses
5. Register Writeback (WB) — update the destination registers

The ideal pipeline in the 486 is shown in the upper table on slide 19 and the ideal pipeline in the Pentium is shown in the lower table. In the ideal case, the Pentium has $CPI^{ideal} = 0.5$.

The ideal case for multiple pipelines is difficult to achieve, for two reasons:

1. Dependent instructions cannot be paired. In the program example on slide 20, I_0 depends on the result of I_1 and so they cannot be executed in parallel. Instead, I_1 is stalled in stage ID1 for an extra CC until the result for EAX is available.
2. If the first instruction of a pair is stalled, the second instruction must stall to prevent it writing before the first (WAW hazard). In the Pentium, some registers (such as the stack pointer ESP) are not forwarded, causing implicit hazards on stack operations (push/pop).

Slide 21 Instruction Pool Example

A superscalar with replicated EUs is simpler to implement. This strategy was chosen for Intel x86 processors after the Pentium (beginning with the Pentium Pro, Pentium II, and used to this day).

The example shows a superscalar version of the DLX pipeline and a program fragment. In this example:

1. I_1 — I_3 have no dependencies. I_1 and I_2 are ALU instructions and I_3 is a Load. All three can be run in CC1 — I_1 and I_2 in two EX units and I_3 in a MEM unit.
2. Each of I_4 — I_6 is dependent on the previous instruction and must be run sequentially in CC2 — CC4.
3. I_7 is independent of prior instructions — it could be rescheduled to run in a MEM unit in any of CC1 — CC3.

Most programs consist of sequential steps and contain data dependencies, as seen in I_4 — I_6 . On CC2 — CC4, only one instruction is executed, an inefficient use of the multiple EUs. Finding more instructions, such as I_7 that can be rescheduled to run in these clock cycles improves performance. For this reason, rescheduling is a critical feature of superscalar pipelines.

Slide 22 Instruction Rescheduling Strategies

Strategies for instruction rescheduling must avoid the **hazards** associated with data dependency. The goal of rescheduling is to remove RAW hazards, but the rescheduling must not create new WAR or WAW hazards.

If instruction I_{k+j} depends on the result of instruction I_k then I_{k+j} must wait (not be executed) until I_k completes. If this wait causes a pipeline delay then the delay is called a **stall**. If other useful work can be rescheduled into the waiting time, then the CPU is kept busy — this is called **hiding the latency**.

Some data dependencies are caused by the compiler reusing the same register for successive **unrelated** instructions in the high level code. For example, the compiler might use R1 as a load and ALU register in the instructions

x = x + 4

y = y + 5 .

This creates a dependency through R1 even though x and y are independent. This is called a **false dependency** and can be repaired by **register renaming**. The compiler can use R1 for the x instruction and R2 for the y instruction — there is no longer any dependency in the compiled code.

In general, instruction rescheduling can be described as increasing the "distance" between dependent instructions I_k and I_{k+j} — hiding the latency by rescheduling unrelated useful work between the dependent instructions. There are two basic methods of rescheduling:

1. **Compiler optimization** of the object code in the second pass,
2. **Dynamic rescheduling** (out-of-order execution) of the machine code by the CPU at run time.

Slide 23 Efficient Register Reuse

A basic compiler strategy is efficient reuse of registers (without consideration of rescheduling). Registers are allocated to loaded memory operands, and program efficiency often depends on de-allocating (retiring) registers as soon as they are not required, so they can be reused for new instructions.

In the program fragment, R1 is **allocated** in the LW instruction. LW overwrites R1, so that any value assigned by a previous program fragment is lost.

The ADDI instruction reads and writes R1 and an operand. The SW instruction reads R1 as an operand and stores its value. The store instruction is the last use of R1 in this instruction block. It will not be used again as a source in the instruction block. This signifies the **de-allocation** of R1 for the instruction block.

The fourth instruction (ADD) again overwrites R1. The old value of R1 is lost and R1 is now **re-allocated**.

Slide 24 – 26 Register Reuse versus Register Renaming

When rescheduling is performed, the efficient register **reuse** is replaced by register **renaming**. Instead of re-allocating a retired register, a new register is allocated to avoid false dependencies.

In the first pass of the compiler on the program fragment, the last instruction (ADD) re-allocates R1. In a second optimization pass, the register is **renamed**. This means that instead of re-allocating R1 the compiler allocates R10. Then the renamed instruction can be **rescheduled** to follow the first load. Since the DLX requires a stall cycle between LW and ADDI, this rescheduling removes the stall by performing useful work (ADD) while the LW is loading the operand. It is said that this rescheduling hides the latency.

The example on slide 25 shows three lines of high level code and the first pass output of the compiler. R2 is allocated and re-allocated to variables X, Y, Z. This code cannot be rescheduled. In the second pass, the compiler performs register renaming — it allocates R3 to Y and R4 to Z. The resulting code can be rescheduled because the three program blocks (corresponding to the three high level instructions) are independent.

Comparing the program execution on a scalar (single pipeline DLX), the rescheduled code runs in 13 CC instead of 16 CC — a 23% improvement in performance (slide 26).

Slide 27 – 29 Scheduling Algorithm — Instruction Scoreboard

The instruction **scoreboard** method permits detailed control over instruction execution order, taking account of data dependencies and systems policies. Each decoded instruction in the instruction pool is assigned a status field. The DLX requires four status values:

1. **NR** (Not Ready) — A source operand is not available so the instruction is not ready
2. **R** (Ready) — All source operands are available and the instruction can be executed
3. **X** (Executed) — The instruction was executed in the ALU but the result is not available
4. **F** (Finished) — The Instruction was executed and the result is available

Only instructions marked **R** can be executed. Initialization instructions in any program must have status R for any legal program. Execution order also depends on system policy.

The scoreboard table shows the program listing with instructions initially marked NR or R. The table is updated on each clock cycle, as an R instruction is executed. The columns show the CC in which the instruction finishes in the EX execution stage (ALU).

Most instructions are marked F when they complete EX. LW instructions load the operand in stage MEM (after EX) and so are marked X — in the following CC they are marked F.

Slide 28 shows the scoreboard for the renamed (but not rescheduled) code from slide 25. In this example, the policy is that instruction execute in original program order. At program start, each LW is ready. Clock cycles are counted so that the first LW enters the ALU and is marked X in CC1. No other instructions are affected. In CC2, the first LW is marked F, so that the first ADD is marked R (its source operands are now available). In CC3 ADD is executed and marked F (it does not need the extra CC that LW needs), so that SW can be marked R. This pattern continues until the last SW. On CC1, CC5, CC9, no instruction is marked F — this indicates the pipeline stall caused by the Load-ALU hazard.

Slide 29 shows the scoreboard for the same code, but this time the policy is that instructions are executed in the order they become Ready. At program start, each LW is ready. The first LW is marked X in CC1, the second LW is marked X in CC2, and the third LW is marked X in CC3. In CC2, the first LW was marked F, so the first ADD is marked R — this add is executed and marked F in CC4. Similarly, the SUB was marked R in CC3 and is executed and marked F in CC5. This pattern continues until the last SW. An instruction is executed (to X or F) on every clock cycle, so there are no stall cycles. The order of instruction execution is the order of the rescheduled code from slide 25.

Slide 30 – 31 Virtual Register Architecture

The registers visible to the programmer at the level of ISA semantics are called **Architectural Registers** (AR). The DLX was defined with 32 integer ARs, R0 – R31.

Most modern CPUs have additional **Virtual Registers** (VR) not visible to the programmer. The VRs are n real physical registers **VR0, VR1, ... , VRn-1** hidden in the instruction pool. Generally, the VR set is larger than the AR set. The Pentium 4 has 8 ARs compared to 128 VRs.

The CPU assigns a VR to each AR found in an instruction block. This way, register renaming in the compiler is not necessary. If two unrelated blocks both use R1, then one block will be assigned VR1 ← R1 and the second block will be assigned VR2 ← R1. This is a form of **automatic register renaming** in the CPU hardware.

Slide 31 shows an example with 2 instruction blocks. In the first block, the machine instruction allocates VR1 to R1 by loading an operand from memory, and then allocates VR2 to R1 by updating the register. The instructions saved in the instruction pool do not refer to R1 — they are recoded using VR1 and VR2. This instruction block accesses VR1 as destination and source operands until R1 is re-allocated. Stable (unchanging) registers (R2 in the example) do not require virtual registers — the decoded instruction in the instruction pool is written using the literal value read from the AR. When the next block re-allocates R1, the decoder allocates VR3 to R1, and this block uses VR3 as its operand.

VRs can be **retired** (de-allocated) when they are no longer the source for any operation in a block. The de-allocation of VRs requires copying the VR value to the original AR (commitment to state). Commitment must be performed in the correct program order to prevent WAW and WAR hazards. Thus, the first block must write VR2 to R1 before the next block writes VR3 to R1.

Slide 32 – 33 Bookkeeping — Register Allocation Table (RAT)

In order to keep track of the VR allocations, the instruction pool includes a **Register Allocation Table** (RAT). The RAT can be implemented in several ways.

Since there are more VRs than ARs, an intuitive approach is a RAT indexed by the VR number. The entry describes the AR for which the VR is allocated, an instruction block ID, and the VR status (allocated or available).

On Intel processors, the RAT index is the AR name, so that each index can have multiple entries. An entry describes the allocated VR, a sequence number that keeps the allocated VRs in chronological order, and a status field that marks the AR as stable or allocated (currently assigned as a destination to an unfinished instruction in the instruction pool).

When an instruction is decoded in an Intel CPU,

1. The name of each stable source AR is replaced by the value stored in the AR.
2. The name of each allocated source AR is replaced by the VR with the highest sequence number in the RAT for that AR. Thus, in the example on slide 31, the processor must commit VR1 to R1 before committing VR2 to R1 (or it may just skip committing VR1).
3. The name of each destination AR is replaced by a new VR allocation, and a new RAT entry is written. Available VRs are found by searching the RAT for unallocated VRs. The decoder allocates available VRs sequentially mod n, in order to maximize the code distance between VR reuse.
4. The recoded instruction (with literals and VRs) is placed in the instruction pool. Instructions are executed from the pool by reading and writing VRs.

A VR is retired on either of two conditions:

1. An instruction that overwrites the allocated AR is marked finished.
2. The VR does not appear as a source for any unfinished instructions in the pool. In this case, the VR (with the highest sequence number) is copied to the AR and the AR is marked stable in the RAT. The next use of the AR as a source will access the stable AR value.

Slide 34 **Virtual Register Example**

I1 allocates VR1 to R1

I2 allocates VR2 to R2

reads VR1 as the source for R1

I3 reads VR2 as the source for R2

I4 overwrites R2 $\Rightarrow$ retire VR2

allocates VR3 to R2

reads stable register values as the sources for R3 and R4

I5 allocates VR4 to R3

reads VR3 as the source for R2

I6 overwrites R2 $\Rightarrow$ retire VR3

allocates VR5 to R2

reads stable register value for R4

I1 and **I2** have completed and been marked finished. VR1 is no longer a destination register in the instruction pool and can be retired and committed. VR1 is copied to R1. R1 is marked stable.

I7 allocates VR6 to R4

reads stable register value for R1

reads VR5 for R2

Slide 35 – 36 WAR and WAW Pseudo-Dependencies

The code fragment on slide 35 contains dependencies through R1. Since I2 and I3 write to R1, all three instructions are actually independent. This is called **pseudo-dependency**. Like real data dependency, the instructions cannot be rescheduled:

There is a WAR hazard if I1 is placed after I2 or I3.

There is a WAW hazard if I2 is placed after I3.

The pseudo-dependency can be removed using virtual registers:

In I1, allocate VR1 to R2 and read stable register values from R1, R3

In I2, allocate VR2 to R1 read stable values from R3, R4

In I3, allocate VR3 to R1 read stable values from R3, R5

The recoded instructions are

I1 ADD VR1, R1, R3

I2 ADD VR2, R3, R4

I3 ADD VR3, R3, R5

The recoded instructions can be executed in any order. But the virtual registers VR1, VR2, and VR3 must be retired in program order. If I3 is in the instruction pool when VR2 is retired, the value of VR2 will not be committed to R1 (only the last write to a VR associated with R1 is committed).

In a more realistic code fragment on slide 36, the red and green sequential code sections have pseudo-dependencies through R1. The code must run sequentially over 7 clock cycles on a superscalar CPU with multiple ALU and memory EUs. After automatic register renaming with virtual registers, the two code sections can be efficiently rescheduled to run in 4 clock cycles, a 75% improvement in performance.

Slide 37 Dynamic Scheduling

Rescheduling can be performed by the compiler or by the CPU. When the compiler schedules the code, it can only refer to the limited set architectural registers. The CPU can take advantage of the larger set of virtual registers to achieve a better optimization. Rescheduling in the CPU is called **dynamic scheduling**, first introduced in the Tomasulo algorithm for the FPU in IBM mainframes (1967).

Modern dynamic scheduling is based on the superscalar CPU with a fast decoder, automatic register renaming (with VRs), and a scoreboard that marks instructions in the pool with a status field — not ready, ready, finished.

Instructions marked ready in the instruction pool are **issued** (sent to an EU for execution), returned to the pool with the instruction result in the destination VR, and marked finished. Finished instructions are retired in original program order, and their VRs are committed to state.

Slide 38 – 45 Tomasulo Algorithm

The Tomasulo algorithm was the first step in dynamic scheduling. Operation was similar to modern dynamic scheduling, except that there was no RAT and instructions were committed immediately after execution, not in program order.

In the Tomasulo algorithm the instruction pool is called a **reservation station** (RS). Each decoded instruction is buffered in an RS until it is ready for execution. The number of the RS is used as a VR to identify the destination register for that instruction. Source values (literals) from stable registers are encoded into the instruction. Source values from an unfinished instruction are allocated the RS number (as a VR) that points to the instruction that will provide the value.

Ready instructions are moved to an appropriate EU for execution. Immediately after execution, the RS number and result are broadcast on the **Common Data Bus** (CDB). All instructions waiting for that result update their source VRs with the numerical result and store instructions to data memory are completed.

Slides 40 – 43 show a sample program in execution. Slide 40 shows the decoded program, which can run in 3 clock cycles. Slides 41 – 43 show the scoreboard after CC1, CC2, and CC3.

On CC1, three instructions with stable source operands are marked finished. Two instructions that use the destinations of these three are now ready and execute in CC2. On CC3 the VR3 and VR7 are updated. As soon as the value of VR is broadcast on the CDB, the SW instruction can be completed (store operations can go directly from NR to F).

The Common Data Bus provides a simple distributed control mechanism. There is no central scoreboard controller or RAT. This simplicity causes certain problems. The CDB becomes a bottleneck, limiting the performance of other subsystems. Also, instructions are retired by execution order, not program order. This means that branch instructions must be completed before any following instructions are executed (this issue is discussed in presentation 4). Also, there is no well-defined break point for exceptions.

Slide 46 In-Order Retirement

The update of the Tomasulo algorithm is **in-order retirement**, which commits results to architectural state in the order of the original program. The retirement order is set at instruction decode time by the value of the source PC for the instruction. The PC of the last retired instruction is the **progress index** for the program. Instructions beyond the progress index remain in the instruction pool until all instructions with a lower PC are retired.

In-order retirement permits **speculative execution**. This means that it is possible to schedule the execution of instructions that may never run, for example instructions following the target of a conditional branch. If the branch evaluates as taken, then the execution of these instructions turns out to be useful work. But if the branch evaluates as not taken, then these finished speculative instructions must be canceled and deleted from the instruction pool. Speculative execution includes **branch prediction** — the CPU guesses whether a branch is taken or not taken and corrects the instruction pool if the branch evaluates differently (**misprediction**).

In-order retirement also permits **precise exceptions**. If an instruction with PC_n raises an exception that changes the program order, then the finished instructions in the pool with $PC > PC_n$ are canceled and deleted. The exception is handled for PC_n and the subsequent instructions are decoded and executed after the exception is handled.

Instructions are moved from the instruction pool to the **retirement queue** in the order of PC, if they are marked as finished and valid. Valid means, no errors, exceptions or mispredictions.

Slide 47 – 48 Dynamic Scheduling — Tomasulo + In-Order Retirement

Many advanced RISC processors, including IBM Power, ARM, and Sparc CPUs, perform dynamic scheduling. All x86 CPUs since the Pentium Pro (1995) use dynamic scheduling, also called out-of-order execution.

Because the instruction pool permits dynamic scheduling, it is called a **Reorder Buffer** (ROB). An ROB entry contains the instruction PC, the operation, status, VR pointers, and stable operand values. When all VR pointers for an instruction describe destination operands of finished instructions, the instruction is ready.

Instructions are loaded to the ROB from the decoder. When an instruction is marked ready, it is sent to a reservation station (RS) for execution in an EU. The instruction result replaces the destination VR for the instruction in the ROB, the RAT is updated and the instruction in the ROB is marked as finished. Finished instructions are moved to the retirement unit in program order, according to the instruction PC.

Instructions proceed out-of-order between the ROB and RS. Instructions proceed in-order (FIFO) from the decoded to the ROB to the retirement unit.

All data forwarding (bypass) is performed in the ROB. For the pair of instructions

$\mathbf{I_k}$ `ADD R1, R2, R3`
$\mathbf{I_{k+j}}$ `ADD R4, R5, R1`

the decoding of $\mathbf{I_{k+j}}$ depends on the condition of $\mathbf{I_k}$:

If $\mathbf{I_k}$ is retired, then the value of **R1** is loaded from the architectural register

If $\mathbf{I_k}$ is finished but not retired, then the value of **R1** is loaded from the destination register of $\mathbf{I_k}$

If $\mathbf{I_k}$ is not finished, then the value of **R1** is set to a source VR equal to the destination VR for $\mathbf{I_k}$

Slide 49 – 52 Intel x86 Dynamic Scheduling

Since the Pentium Pro, the **x86 ISA** has had certain common features. Modern x86 CPUs can be thought of as a superscalar RISC core with an adaptation layer (in silicon) for handling the IA-32 CISC instruction set.

The x86 Fetch and Decode stages convert IA-32 instructions to 1 – 6 μops per CC. These micro-ops are essentially RISC-type instructions. The CPU performs dynamic scheduling with an ROB that holds micro-ops until ready for execution. A scheduler looks for ready micro-ops and issues micro-ops to parallel execution in multiple EUs — ALU, FPU, Load, Store. Finished micro-ops update execution results to instruction pool. Retirement (in-order write back) is performed by PC order so that finished micro-ops commit to architectural state in original program order.

Slide 50 shows an example of three IA-32 assembly language instructions located in instruction memory (cache). Although IA-32 ARs are named EAX, EBX, and so on, the decoder allocates VRs

R1 — R7. The 3 machine language instructions are decoded to 9 micro-ops and placed into the ROB.

Execution order is according to the order that instructions become ready, but limited by available EUs to 1 load and 1 store per clock cycle. Slide 51 shows the scoreboard for program execution:

Initially all the LW instructions are ready, but they must execute sequentially because there is only one load unit.

On CC1, the first LW is executed and marked F. The ADD is now marked ready.

On CC2, the second LW and the ADD execute in parallel. Now the first SW and first SUB are marked ready.

On CC3, the first SW, first SUB, and last LW execute in parallel. The second SW and second SUB are marked ready.

On CC4, the second SW and second SUB execute in parallel. The third SW is marked ready.

On CC5, the third SW is finished.

Slide 52 shows the organization of the **Nahalem Micro-Architecture**, the version of the x86 pipeline found in one core of an i7 CPU.

At the top in purple is L1 instruction cache and the IF unit. Instructions are fed to the decoders shown in orange. The decoded micro-ops are sent to the 28 Entry LSD buffer. The LSD is a loop stream detector, an example of a technique called **trace cache** (discussed in presentation 5). The LSD detects loops in programs and stores the sequence of micro-ops in each loop — this eliminates the need to decode the loop separately on each iteration, improving loop performance.

Micro-ops are then moved to the RAT for VR allocation and then to the ROB and reservation stations in yellow. The EUs are shown in blue (integer, FP, and SSE vector units) and green (memory access units and data cache). The memory ordering buffer (MOB) is another optimization. Many instructions depend on data loads to proceed, but data stores do not affect performance. The MOB reorders data loads before stores so the program can continue quickly. If a load refers to a location that has not yet been updated by a waiting store, then the MOB provides the temporary value for that location from the queue of store instructions.

The retirement unit is seen in gray to the right of the ROB.

Slide 53 – 54 Deep Superpipeline

The organization of the Nahalem Micro-Architecture on slide 52 is an implementation of the idealized pipeline on slide 49. The number of pipeline stages in this complex implementation is a highly technical question depending on detailed analysis of the number of steps in the operation of each subsystem. The Pentium III was reported to be organized into a 10 stage and the Pentium 4 reported to have a 20 stage pipeline. This is an example of superpipelining (first discussed in slides 42 – 43 of presentation 1).

The general idea of **deep superpipelining** is to divide each pipeline stage in half, so that each new stage performs half the work in half the time.

For the 5 stage DLX pipeline with stages IF, ID, EX, MEM, WB, superpipelining would produce a 10 stage pipeline, with stages IF1, IF2, ID1, ID2, EX1, EX2, MEM1, MEM2, WB1, WB2. Although the pair IF1 , IF2 performs the same work performed in the original IF, the diagram shows that

instruction I_{k+1} is (partially) fetched when instruction I_k moves from IF1 to IF2. This means that

$$\tau_{\text{superpipeline}} = \frac{\tau_{\text{pipeline}}}{2} \quad \Rightarrow \quad \text{Run time}_{\text{superpipeline}} = \frac{1}{2}\text{Run time}_{\text{pipeline}}$$

The problem with deep superpipelining is that not all instructions can be effectively split in half. Some operations require a fixed time and do not scale with clock rate. For example, exceptions such as page faults, branch penalties, and cache updates depend on latency in external devices. If the clock runs faster then more stall cycles will be counted while waiting for the device, and so CPI^{stall} will be larger.

In the transition from the 10 stage pipeline of Pentium III to the 20 stage pipeline of Pentium 4, this caused non-scaling latency a problem for early models. Based on the CPU equation, a 1.5 GHz processor should be 50% faster than a similar processor at 1.0 GHz. That is, if the CPI and IC are about the same for two CPUs then

$$S = T / T' \approx \tau / \tau' = R' / R = 1.5 \text{ GHz} / 1.0 \text{ GHz} = 1.5$$

But comparisons of SPEC Cint on two models found that the 1.5 GHz Pentium 4 (P4) is only 20% faster than 1.0 GHz Pentium III (PIII). Assuming that compilation for the two processors gives a similar IC, one finds that $CPI^{P4} = 1.25\ CPI^{PIII}$, so that it takes 25% more CC to run a program on the P4 than on the PIII. Assuming that $CPI^{ideal} = 1$ on both CPUs and $CPI^{stall-PIII} \approx 0.2$, one finds

$$CPI^{stall-P4} = 1.25 \times CPI^{stall-PIII} + 0.25 \approx 0.5 \quad \Rightarrow \quad CPI^{P4} \approx 1.5$$

Intel publications at the time claimed that the "meaning" of clock speed was different for the PIII and P4, and so the 20% improvement was pure improvement. This claim was equivalent to assigning the factor of 1.5 to the meaning of clock speed, instead of discussing CPI and causes of stalls in deep superpipelining. The matter was soon forgotten because the P4 clock speed was quickly raised to about 4 GHz, a speed the PIII structure could never attain.

Slide 55 – 56 Hyper-Threading

To solve the problem of increased pipeline stalls caused by superpipelining, Intel added **Hyper-Threading**® to the Pentium 4. A Hyper-Threaded processor contains a standard single core pipeline with two separate copies of the architectural registers (architectural state). Since the CPU has two separate program counters, it can fetch two separate program threads at the same time. To the OS, the processor looks like two processors: CPU0 and CPU1.

Since the processor has one execution core (EUs and data memory) it shares the threads by interleaving — it issues instructions (fetches and decodes instructions to the ROB) from alternate threads on alternate clock cycles. Micro-ops in the ROB are labeled by thread and (since the threads are independent) provide increased opportunities for dynamic scheduling.

With Hyper-Threading, if one thread stalls the other thread can continue running until the stall is resolved. CPU 0 and CPU1 keep busy most of the time.

The improvement from Hyper-Threading can be estimated by writing

$$CPI^{stall} = CPS \times P_s$$

where CPS = cycles per stall, and P_s is the probability of a stall per instruction.

With Hyper-Threading, the CPU only suffers a stall if both threads stall at the same time. Since the threads are independent, $P(\text{2 simultaneous stalls}) \approx (P_s)^2$ and $CPI^{stall\text{-}HT} \approx CPS \times (P_s)^2$. Assuming that each stall is 2 clock cycles, so $P_s = 0.5 / 2 = 0.25$, the speedup is found from

$$S = \frac{CPI \times IC \times \tau}{CPI^{HT} \times IC^{HT} \times \tau^{HT}} = \frac{CPI}{CPI^{HT}} = \frac{1 + CPS \times P_S}{1 + CPS \times \left[P_S\right]^2} = \frac{1 + 2 \times 0.25}{1 + 2 \times \left[0.25\right]^2} = 1.2$$

This 20% speedup is precisely the value published by Intel (slide 27 in presentation 7).

Slide 57 Hardware Utilization in x86 Example

The program on slide 51 executed in 5 clock cycles, instead of 9 clock cycles (if each instruction executed sequentially). The speedup of 9/5 = 1.8 is significant. But most of the execution units were idle most of the time. Higher performance requires more instructions scheduled to run in parallel. Increasing instruction level parallelism (ILP) requires attention to general program structure.

Slide 58 – 60 Overhead in Performance Calculations

A typical program section consists of:

1. **Head** — set up instruction that run one time
2. **Main functions** — program loops that run over many iterations
3. **Tail** — shut down instructions that run one time

The head and tail contribute a small **overhead** to the main functions that require most run time.

Similarly, the execution of a program in a superscalar consists of overlapping activities:

1. The Fetch + Decode units run in the **background** to keep the ROB filled. Because they are pipelined, after the first few CC required to fill the ROB, these units contribute a small **overhead** to run time.
2. The **program executes** on a set of u EUs (EU_1, EU_2, ... , EU_u). The larger IC, the longer the run time in the EUs. The more EUs being utilized, the shorter the run time.
3. The **retirement unit** removes instructions from the ROB. Because it is pipelined, after the first few CC, this unit contributes a small **overhead** to run time.

For example, a program might require 8 CC to fill the ROB, 10^6 CC to execute a loop (while IF+ID continuing filling the ROB and the retire unit removes completed instructions in the background), and another 8 CC to finish emptying the ROB after the main loop completes. The total number of CC required to run a program can be split into

$$N = N_{IF+ID} + N_{program} + N_{retirement} = N_{IF+ID} + N_{head} + N_{main\ loops} + N_{tail} + N_{retirement} = N_{overhead} + N_{main\ loops}$$

$$\text{where } N_{overhead} = N_{IF+ID} + N_{head} + N_{tail} + N_{retirement}$$

For a simple program with one loop, with $N_{one\text{-}loop}$ clock cycles per loop iteration and $n_{iterations}$ iterations,

$$N_{main\ loop} = n_{iterations} \times N_{one\text{-}loop}$$

In this approximation, the CPI can be estimated as

$$CPI = \frac{N_{overhead} + n_{iterations} \times N_{one\text{-}loop}}{IC_{overhead} + n_{iterations} \times IC_{one\text{-}loop}} = \frac{\frac{N_{overhead}}{n_{iterations}} + N_{one\text{-}loop}}{\frac{IC_{overhead}}{n_{iterations}} + IC_{one\text{-}loop}} \xrightarrow[n_{iterations} \to large]{} \frac{N_{one\text{-}loop}}{IC_{one\text{-}loop}} = CPI_{one\text{-}loop}$$

In the ideal case, dividing the work among u EUs divides the number of clock cycles by u, and so reduces CPI to

$$N_{one\text{-}loop} = \frac{N^{u=1}_{one\text{-}loop}}{u} \Rightarrow CPI = \frac{1}{u} \times \frac{N^{u=1}_{one\text{-}loop}}{IC_{one\text{-}loop}} = \frac{CPI^{u=1}_{ideal}}{u} = \frac{1}{u}$$

The example on slide 60 shows a program with data dependencies in R1 and R2, so that only SW and BNEZ can be run in parallel. The ADDI instruction only runs once and is overhead. In each loop iteration, 5 instructions run during 4 clock cycles.

The loop runs over 0x1000 / 4 = 0x400 = 1024 iterations. Including the IF+ID and retirement overhead, and ADDI

$$CPI = \frac{overhead + 4 \times 1024}{1 + 5 \times 1024} = \frac{\frac{overhead}{1024} + 4}{\frac{1}{1024} + 5} \to 0.80$$

Clearly, the average CPI for the program can be approximated by the CPI for one loop.

Slide 61 – 69 Loop Unrolling

One method to increase instruction level parallelism (ILP) is called **loop unrolling** — combining two or more loop iterations into one. With multiple EUs and automatic register renaming, it is often possible to improve performance by running multiple loop iterations at the same time.

The program on slide 60 (and on the left side of slide 61) must run sequentially because of the data dependencies through R1 and R2. On slide 60, the CPI was found to be 0.80 on the superscalar. With register renaming (performed automatically in the CPU or by the compiler as an optimization) the loop can be unrolled to perform 4 loop iterations at one time.

The scoreboard for the unrolled program on the 2-ALU superscalar is shown on slide 62. In the unrolled program, 14 instructions complete in 7 CC, so CPI = 0.50 . This represents an improvement of S = 0.8 / 0.5 = 1.6 over the initial version. The scoreboard for the same program on a CPU with 4 ALUs + 4 Read Units + 4 Write Units is shown on slide 63. The CPI is now 0.29 with an improvement of S = 0.8 / 0.29 = 2.8 over the original version.

Slide 64 shows a code fragment written in x86 assembly language. This program loads data from memory, adds a table entry to the operand and stores the result. Slide 65 shows the VR allocation and recoded instructions for the code fragment. The loop unrolling for this program is accomplished in the CPU after decoding to micro-ops. The table index ESI that runs from 1 to 400 in the original version is split into VR30 that runs from 1 to 200, and VR31 that runs from 201 to 400. The initial decoding is shown on the left side of slide 67, and the unrolled decoding is shown on the right side. On the right side, 4 instructions are doubled. The scoreboard on slide 68 shows that one loop iteration for the original version takes 4 clock cycles. The program runs sequentially, with low utilization of the EUs in the superscalar. The scoreboard on slide 69 shows that one loop iteration for the unrolled version also takes 4 clock cycles. The program runs with higher utilization of the EUs in the superscalar, executing 4 instructions per clock cycle. The unrolled version runs twice as fast because the number of loop iterations is divided in half.

Slide 70 – 76 Software Pipelining with Explicit Parallelism

Some modern CPUs, such as the Intel 64-bit Itanium (not an x86 processor), permit the compiler to indicate parallel code blocks explicitly. In assembly language, the end of a parallel block is indicated by the symbol **;;** which tells the CPU that instructions up to **;;** are marked ready and instructions after **;;** are marked not ready. This facility permits a method of software unrolling called **software pipelining**.

Conventional code appears at the top of slide 70. Every instruction executes sequentially on the superscalar because of data dependencies. Below appears a software pipelined version. The header prepares all sources required for the first loop iteration. In loop iteration N, the loop code reads only sources prepared in iteration N – 1 and prepares all of the destinations that will be read as sources in iteration N + 1.

The left side of slide 71 shows a sequential code fragment and the pipelined version appears to the right. Most instructions in the header run sequentially. All instructions in the loop run in parallel. After the final loop iteration, the tail completes the storage of the last results.

Slides 72 and 73 show the register values while running the header and first two loop iterations. Each source is the last value set for that operand. Slide 74 shows the complete run if the original loop had 3 iterations.

The unrolled software pipelined code appears on slide 75. The scoreboard is shown on slide 76, and as expected the entire loop of 14 instructions runs in 1 clock cycle, so CPI = 0.07, an improvement of 0.80 / 0.07 = 11 over the original version and an improvement of 0.50 / 0.07 = 7 over standard loop unrolling.

Slide 76 Improving Hardware Utilization — Additional Directions

Loop unrolling can improve hardware utilization under appropriate conditions (in particular, independence between iterations).

When a single program thread is highly sequential and with tight data dependencies (for example, bubble sort), finding independent instructions to keep the EUs busy can be difficult.

Instead of parallelizing a single loop-oriented procedure, another source of independent instructions is **speculation**. In speculative execution, **branch prediction** is used to find branch operations that are likely to execute (for example a conditional function call in a high level program). With dynamic scheduling, it is possible to execute instructions that follow the branch and save them in the ROB until the branch is evaluated completely. If the prediction is correct, then much work has already been finished. On a misprediction, the instructions in the ROB are cancelled.