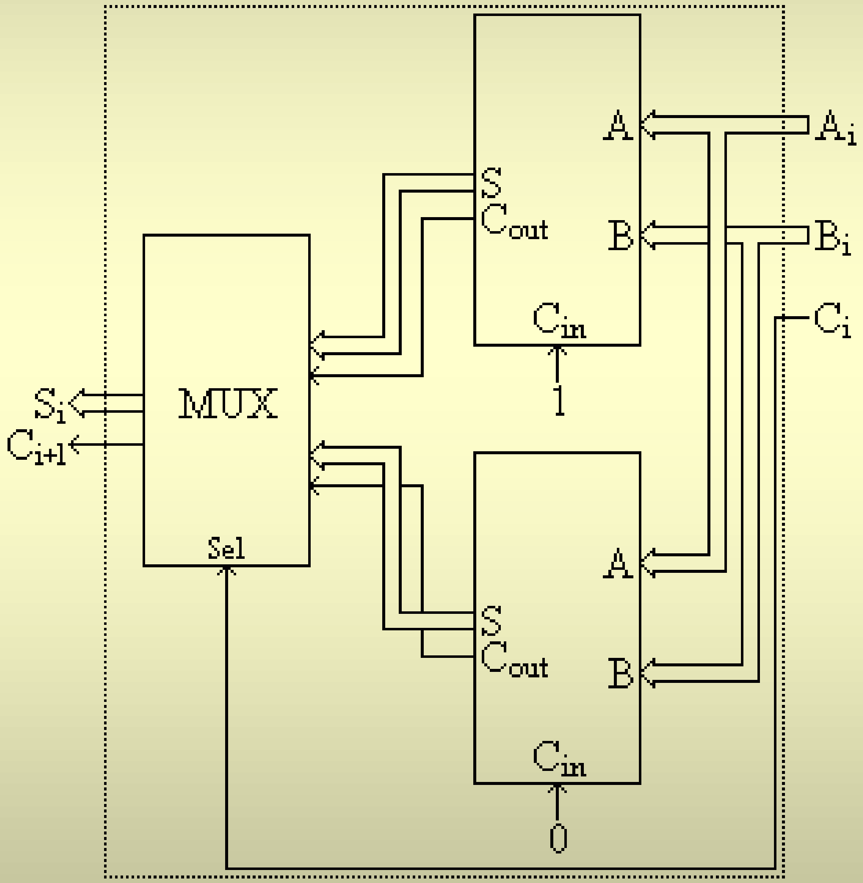
II.4. Speculative Execution

Speculative Execution

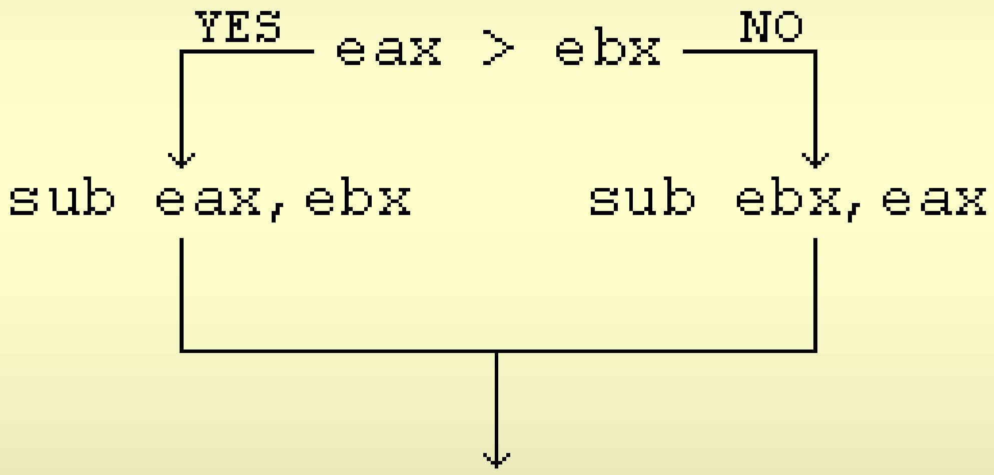
- related to branch prediction
- all possible variants are executed
  - before knowing which is the correct one
- when the correct variant is known, its results are validated
- can be used in simple circuits as well

Example - Selection Adder



How Does It Work? (1)

- conditional jump instructions



- both variants are executed in parallel
- how are the registers (`eax`, `ebx`) modified?

How Does It Work? (2)

- none of the variants modifies them
- the subtraction results are written into temporary registers
- when the relation between `eax` and `ebx` is known
  - determine the correct execution variant
  - update the values of `eax` of `ebx` according to the results obtained in the correct variant

Speculative Execution vs. Prediction

- no wrong predictions
  - success rate - 100%
- requires a lot of registers to keep the temporary results
- managing those registers - difficult
- each execution variant may contain other jumps etc.
  - variants are multiplying exponentially

II.5. Predication

Predication (1)

- used by the Intel IA-64 architecture
  - and in other processing units as well
- similar to speculative execution
- the processor contains predicate registers
  - predicate - Boolean condition (bit)
- each "normal" instruction has associated such a predicate

Predication (2)

- an instruction produces effects if and only if the associated predicate is *true*
  - otherwise the result is not written to its destination
- test instructions can modify predicate values
- this way, program branches can be implemented

Example

- pseudocode

```
if(R1==0){
  R2=5;
  R3=8;
  }
else
  R2=21;
```

Example (continued)

- "classic" assembly language

```
    cmp R1,0
    jne E1
    mov R2,5
    mov R3,8
    jmp E2
E1: mov R2,21
E2:
```

Example (continued)

- predicate assembly language

```
      cmp R1,0,P1
<P1>mov R2,5
<P1>mov R3,8
<P2>mov R2,21
```

- predicates P1 and P2 work together
  - P2 is always the negation of P1
  - first instruction changes both P1 and P2

II.6. Out-of-order Execution

Out-of-order Execution

- instructions do not terminate necessarily in the same order they began their execution
- purpose - eliminating some pipeline stalls
- possible when there are no dependencies between the instructions

Example

```
in al,278
add bl,al
mov edx,[ebp+8]
```

- first instruction - very slow
- second instruction must wait until the first terminates
- third instruction does not depend on the previous ones - may terminate before them

II.7. Register Renaming

Data Dependencies

- occur when two instructions use the same resource (variable/register)
- only when at least one of the instructions modifies that resource
- if resources are registers, some dependencies can be solved through renaming

Types of Data Dependencies

- RAW (*read after write*)
  - first instruction modifies the resource, the second one reads it
- WAR (*write after read*)
  - the opposite case
- WAW (*write after write*)
  - both instructions modify the resource

RAW Dependencies

- "true" dependencies
- cannot be eliminated

```
mov eax,5
sub ebx,eax
```

- the value written into `eax` by the first instruction is necessary to the next one

WAR Dependencies

- also called anti-dependencies

```
add esi,eax
mov eax,16
sub ebx,eax
```

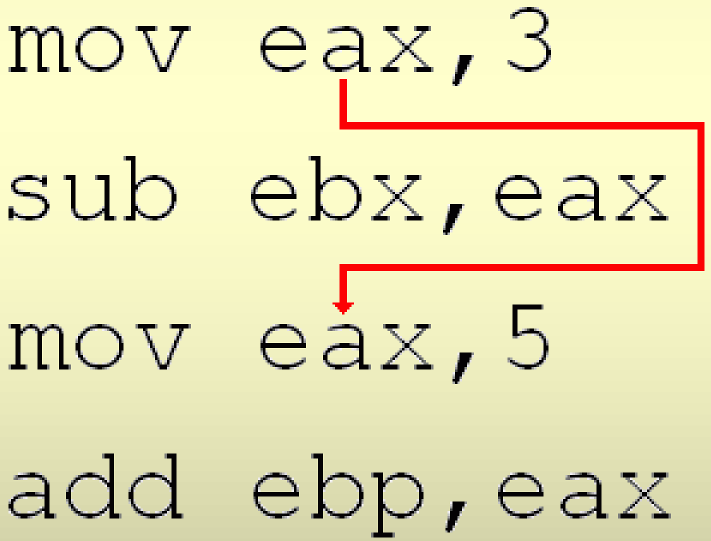
- first instruction must be executed before the second one (not in parallel)

WAR Dependencies - Solution

```
add esi,eax              add esi,eax
mov eax,16   ────────→   mov reg_tmp,16
sub ebx,eax              sub ebx,reg_tmp
```

WAW Dependencies

- also called output dependencies



```
mov eax,3
sub ebx,eax
mov eax,5
add ebp,eax
```

WAW Dependencies - Solution

```
mov eax,3
sub ebx,eax
mov eax,5
add ebp,eax
```

$\longrightarrow$

```
mov eax,3
sub ebx,eax
mov reg_tmp,5
add ebp,reg_tmp
```

Utility

- does renaming help improve the performance?
- increases the degree of parallelization
- more efficient when combined to other techniques
  - superpipeline structure
  - out-of-order execution

Efficiency

- more registers
  - used internally by the processor
  - not accessible to the programmer
- why?
  - renaming is performed automatically
  - the programmer can make mistakes (inefficient resource usage)
  - increase the performance of old programs

II.8. Hyperthreading

Hyperthreading (1)

- a single real (physical) processor, but it is seen as two virtual processors
- duplicates the components that maintain processor status
  - general-purpose registers
  - control registers
  - interrupt controller registers
  - processor status registers

Hyperthreading (2)

- execution resources are not duplicated
  - execution units
  - branch prediction units
  - buses
  - processor control unit
- interlaced execution of the instructions by the virtual processors

Why Hyperthreading?

- the pipeline structure is better exploited
  - when an instruction of a virtual processor stalls, the other processor takes control
- not the same performance gain as in case of a second physical processor
- but complexity and consumption are almost the same as for a single processor
  - there are only few status components

II.9. RISC Architecture

Classic Structure of the CPU

CISC (*Complex Instruction Set Computer*)

- large number of instructions
- very complex instructions $\rightarrow$ long execution times
- small number of registers $\rightarrow$ frequent memory accesses

Practical Observations

- many instructions are rarely used
- 20% of the instructions are executed 80% of the time
  - or 10% are executed 90% of the time
- complex instructions can be simulated through simple instructions

Alternative Structure

RISC (*Reduced Instruction Set Computer*)

- simpler instruction set
  - fewer (relatively) instructions
  - instructions are simpler (elementary)
- large number of registers (tens)
- less memory addressing modes

Memory Access

- fixed instruction format
  - same number of bytes, even though not all are necessary in all cases
  - decoding is simpler
- *load*/*store* architecture
  - memory acces - only transfer instructions (memory $\leftrightarrow$ registers)
  - all other instructions work with registers only

RISC Structure - Advantages

- faster instructions
- reduces the number of memory accesses
  - depending on compilers' capacity to use all registers
- simpler memory accesses
  - less pipeline stalls
- requires less silicon - can integrate additional circuits (e.g., cache)

II.10. Parallel Architectures

Parallel Computing - Usage

- communication between applications
  - concurrent processing can also be used
- higher performance
  - scientific computing
  - very large data amounts to be processed
  - modeling/simulation
    - meteorology, astronomy, etc.

How to Reach Parallelism?

- pipeline structures
  - sequential/parallel
- multiprocessor systems
  - basic units - the processors
- distributed systems
  - basic units - the computers

Performance

- ideal - speed grows linearly to the number of processors
- real - a program cannot be fully parallelized
- examples
  - I/O operations
  - sorting operations

Scalability (1)

- performance growth along with the number of processors
- problems - systems with a very large number of processors
- limitation factors
  - connection complexity
  - time used for communication (overhead)
  - sequential nature of applications

Scalability (2)

- works for a relatively small number of processors
- relatively large number
  - performance growth does not follow processor number growth
- very large number
  - performance stalls or may even drop

Memory Systems

Classifications

- by physical organization
  - centralized
  - distributed
- by access type
  - shared (common)
  - local

Types of Multiprocessor Systems

- symmetric shared memory systems
- distributed shared memory systems
- message-passing systems

Symmetric Shared Memory

- names
  - UMA (*Uniform Memory Access*)
  - SMP (*Symmetrical Multiprocessing*)
- common (shared) memory
- accessible to all processors
- memory access time
  - the same for every processor and every location

Distributed Shared Memory

- names
  - DSM (*Distributed Shared Memory*)
  - NUMA (*Non-Uniform Memory Access*)
- memory is physically distributed
- unique address space
  - visible to all processors
- memory access - non-uniform

Message-passing Systems

- multicomputers
- each processor has its own local memory
  - not accessible to the other processors
- communication between processors
  - explicit messages
  - similar to computer networks