Computer Architectures

D. Pleiter

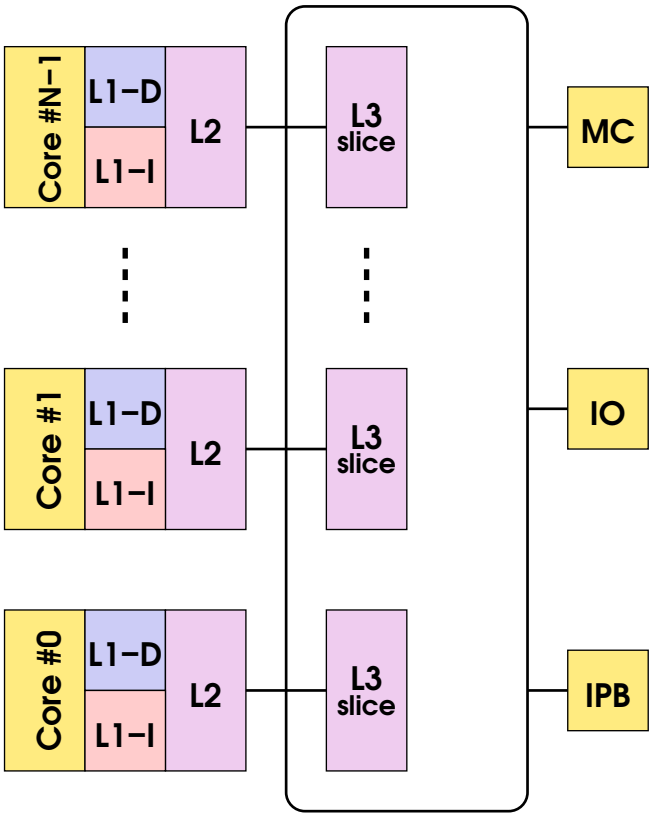
Jülich Supercomputing Centre and University of Regensburg

January 2016

Overview

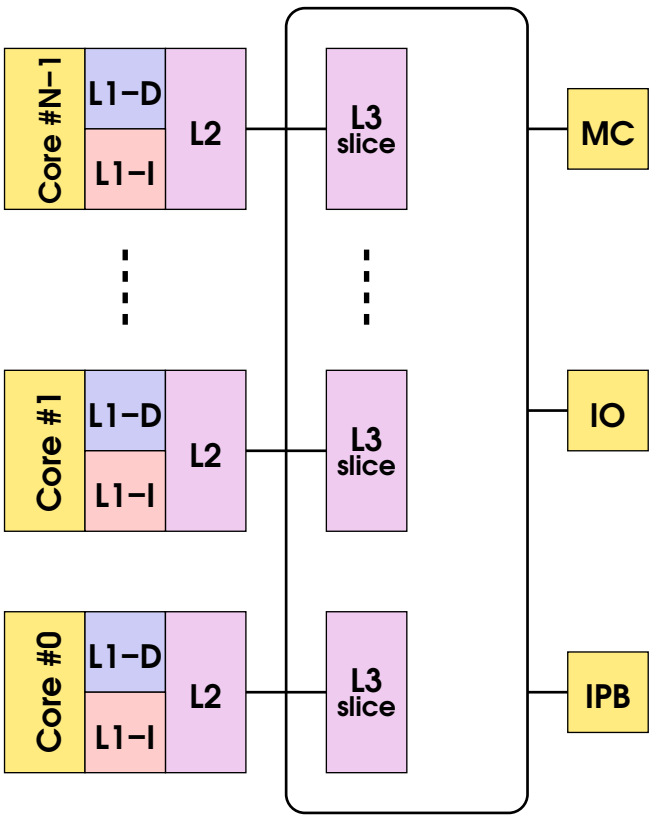
Content

Processor Architecture: Core



- Multi-core architectures have become the default
- Typical components
  - Front-end
    - Instruction fetching
    - Instruction decoding
    - Branch prediction
  - Execution engine
    - Resource allocation
    - Instruction execution
    - Instruction retirement

Processor Architecture: Other



- Memory hierarchy
  - Caches
  - Memory controller
- I/O controller
  - Interface to I/O devices
  - Example: network controller
- Inter-Processor Bus
  - Required for multi-socket designs, only
- Intra-processor network
  - Ring or mesh network
  - Cross-bar switch

Instruction Set Architecture (ISA)

- Relevant ISA categories
  - **CISC**: Complex Instruction Set Computer
  - **RISC**: Reduced Instruction Set Computer
  - **VLIW**: Very Long Instruction Word
- Classification based on internal memory architecture
  - Today's architectures are typically **general-purpose register architectures**
  - **Load-store architectures**
    - Explicit load-store operations only, operands use register addresses only
    - Examples: ARM and POWER ISA
  - **Register-memory architecture**
    - Operand may be in memory
    - Examples: x86 ISA

ISA: Operators

Selected set of categories for instruction operators:

Operator type	Examples
Data transfer	Load and store operations
Arithmetic and logic	Integer arithmetics, bitwise operations, compare operations
Control	Branch, jump, procedure call and return
Floating point	Floating point arithmetics
Mathemetical functions	Inverse, square root

ISA: Memory addressing

Classification according to memory addressing features

- Memory addressing
  - Granularity: Today typically byte addressing
  - Alignment requirements
- Addressing modes (`R9` ... destination register):

Mode	Pseudo-assembler	Description
Register	`mov R0, R9`	Value from registers
Immediate	`mov 0x3, R9`	Constant values
Register indirect	`ld (R1), R9`	Value from location referenced by pointer
Displacement	`ld 0x100(R1), R9`	Pointer plus a fixed offset
Indexed	`ld (R0+R1), R9`	Pointer plus run-time offset
Scaled	`ld 0x100(R0,2), R9`	Pointer plus scaled run-time offset

Introduction to the x86 ISA

- Dominant ISA for desktop computers
- Introduced 1978 ☞ very long-term compatibility
- Meanwhile various variants and extensions:
  - Addition of vector instructions for multimedia applications
  - Extension from 16- to 32- to 64-bit architecture
- General features
  - CISC ISA
  - Register-memory architecture
  - Variable instruction size (8, 16, 24, 32, 40, 48 bits)
  - No data alignment required (few exceptions)

x86 ISA: Data Transfer Instructions

Instructions for explicit transfer of data between memory and registers: `MOV` and variants

Instruction	Description
`mov` *m32*, *r32*	Load 32-bit value to a register
`mov` *r32*, *m32*	Store 32-bit value to memory
`mov` *r32*, *r32*	Move 32-bit value within register file
`mov` *imm*, *r32*	Move 32-bit immediate value to register
`mov` *imm*, *m32*	Move 32-bit immediate value to memory
`mov` *m64*, *r64*	Load 64-bit value to a register
`mov` *r64*, *m64*	Store 64-bit value to memory
`movq` *m64*, *xmm*	Load 128-bit value to a XMM register
`movq` *xmm*, *m64*	Store 128-bit from a XMM register

x86 ISA: Addressing Modes

The x86 ISA supports 7 different addressing modes.

General syntax: *disp*(*base*,*index*,*scale*)

Mode	Example
Absolute	`mov (1024), %eax`
Register indirect	`mov (%ebx), %eax`
Base-indexed	`mov (%ebx,%ecx), %eax`
Based indexed with displacement	`mov 1024(%ebx), %eax`
Based with scaled index	`mov (%ebx,%ecx,2), %eax`
Based with scaled index and displacement	`mov 1024(%ebx,%ecx,2), %eax`

x86 ISA: Arithmetics and Logical Instructions

- **Integer arithmetics**
  - Integer addition (destination is overwritten)

Operation	Example
Add immediate to register content	`add $3, %eax`
Add two register values	`add %ebx, %eax`

  - Similar instructions:
    - Subtraction: `sub`
    - Signed/unsigned multiplication: `imul` / `mul`
    - Integer divide: `idiv`
- **Instructions for bitwise operations**
  - Locigal shift left/right: `shl`, `shr`
  - Arithmetic shift left/right: `sal`, `sar`
  - Bitwise AND/OR/XOR/NOT: `and`, `or`, `xor`, `not`

x86 ISA: Arithmetics and Logical Instructions (2)

- **Comparison instructions**
  - Compare 32-/64-bit values using `cmp` instruction:

Operation	Example
Compare content of register with zero	`cmp %eax, $0`
Compare content of 2 registers	`cmp %eax, %ebx`

  - Operations performed for comparison instruction
    - Subtraction of operands
    - Update status flags in EFLAGS register depending on result
  - Operations performed when using logical compare instruction `test`:
    - Perform bitwise AND of operands
    - Update status flags in EFLAGS register depending on result
  - EFLAGS register used by **conditioned** instructions

x86 ISA: Floating Point Operations

- Floating point addition, subtraction, multiplication and division
- Instructions: `fadd`, `fsub`, `fmul`, `fdiv`
- Operands: memory addresses or FPU stack locations
- Result of operation is stored in FPU register stack
- Stack manipulation
  - Push to stack: `fld`
  - Pop from stack: `pop`
- Example:

Operation	Example
Push value from memory to stack	`fld (%eax)`
Add value from memory to top of stack value	`fadd (%ebx)`
Pop value from stack to memory	`pop (%ecx)`

Processor Core Performance

- Program latency (= **time-to-solution**) may be written as

$$\Delta t = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

- Options to reduce time-to-solution
  - Reduce number of instructions
  - Decrease **cycles per instruction (CPI)**
  - Increase clock rate

Optimising Pipelined Architectures

- **CPI** = Cycles per Instruction
- Assume a pipelined architecture:

$$\begin{aligned}\text{Pipeline CPI} &= \text{Ideal pipeline CPI} + \\ &\quad \text{Structural stalls} + \text{Data hazard stalls} + \\ &\quad \text{Control stalls}\end{aligned}$$

  - **Structural stalls** = Resource conflicts in functional units of a pipeline
  - Possible causes for structural stalls
    - Functional units with different throughput B
    - Memory bus which is used for data and instructions
      ☞ conflicting load operations
- CPI can be reduced by
  - Reducing ideal pipeline CPI
  - Reducing stalls
  - Exploit Instruction Level Parallelism (ILP)

Instruction Level Parallelism (ILP)

- **ILP** = Parallelism among instructions from small code areas which are independent of one another
- Exploitation of ILP
  - Overlapping of instructions in a pipeline
  - Parallel execution of instructions in multiple functional units
  - Vector instructions

ILP: Pipeline

- Consider previous example: $z = a \times b + c - d$

1	`mul R0, R1,` **R5**	store $a \times b$ in register 5
2		stall
3		stall
4		stall
5	`add` **R5**`, R2,` **R6**	add c
6–8	...	stall
9	`sub` **R6**`, R3, R7`	subtract d

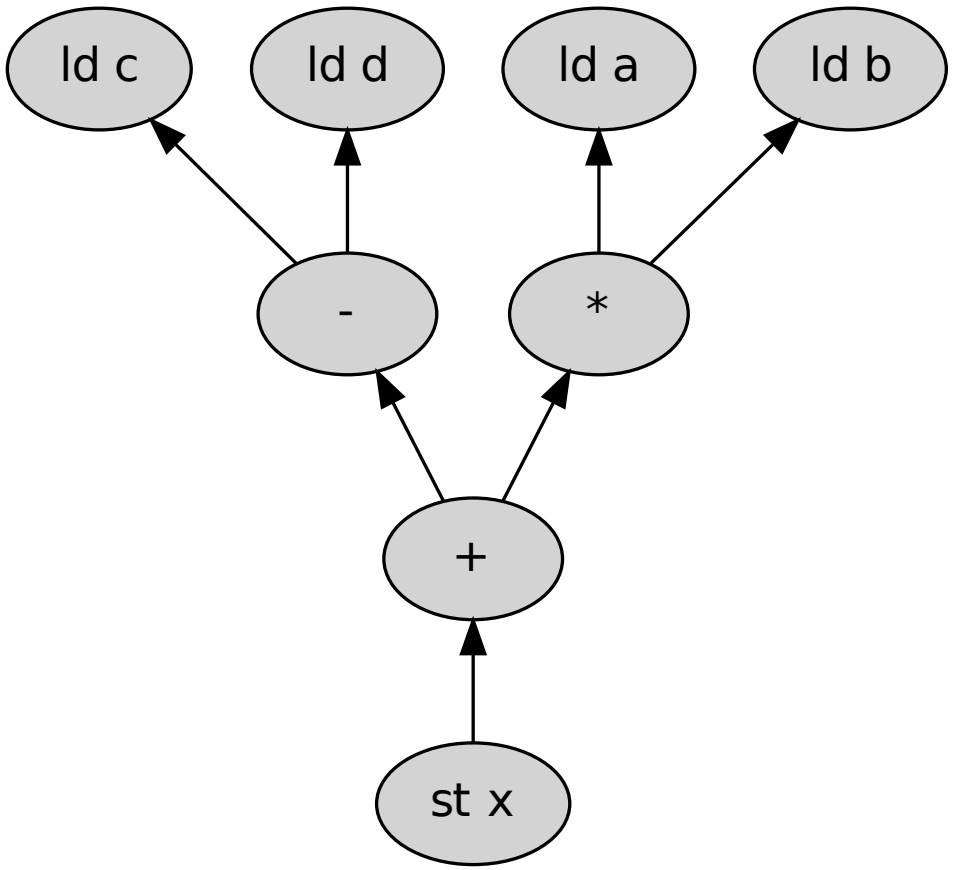
- Re-organisation to improve pipeline filling

1	`mul R0, R1,` **R5**	store $a \times b$ in register 5
2	`sub R2, R3,` **R6**	store $c - d$ in register 6
3		stall
4		stall
5		stall
6	`add` **R5**`,` **R6**`, R7`	add $a \times b$ and $c - d$

ILP: Pipeline (2)

Data dependence diagram for re-ordered schedule

Previously omitted load-store operations included



ILP: Multiple-issue of Instructions

- Allow multiple instructions to be issued within same clock cycle
- Flavours of multiple-issue processors:
  - **Very Long Instruction Word (VLIW) processors**
    - Fixed number of instructions are scheduled per clock cycle
    - Static scheduling at compile time
  - **Statically scheduled superscalar processors**
    - **Superscalar processor** = Processor which can schedule ≥ 1 instructions per clock cycle
    - Variable number of instructions are scheduled per clock cycle and executed **in-order**
  - **Dynamically scheduled superscalar processors**
    - Variable number of instructions are scheduled per clock cycle and executed **out-of-order**

ILP: Vector instructions

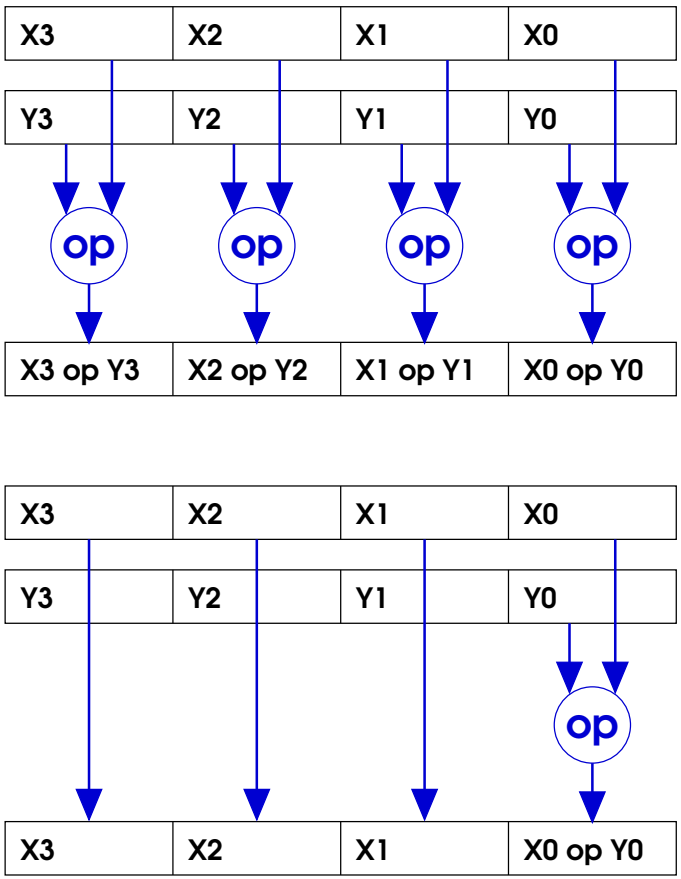
- Vector instructions exploit datalevel parallelism by operating on data items in parallel
  - E.g., vector add

$$\begin{pmatrix} z_0 \\ z_1 \\ z_2 \\ z_3 \end{pmatrix} \leftarrow \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} + \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

- Example ISA:
  - Intel Streaming SIMD Extensions (SSE)
  - Intel Advanced Vector Extensions (AVX)
  - POWER ISA (VMX/AltiVec, VSX)
  - ARMv7 NEON

SSE ISA

- 128-bit wide vector units processing packed or scalar operands
  - Packed operands:
    - 4× 32-bit operands
    - 2× 64-bit operands
- Vector registers `xmm0`, `xmm1`, ..., `xmm15` (number of registers is architecture dependent)



SSE ISA (2)

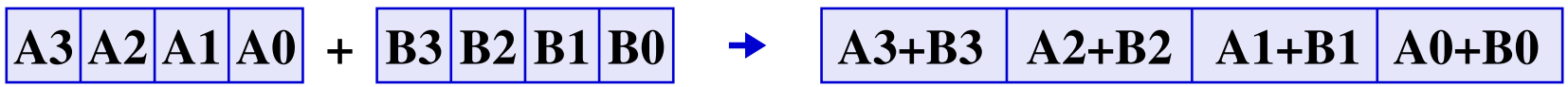
- Typically instructions with 2 operands
  ☞ One of the operands is overwritten
- Types of instructions
  - Load/store instructions
  - Arithmetic and bitwise instructions
  - Shuffle and unpack instructions
  - Cache control instructions
- Multiple generation of SSE extensions: SSE, SSE2, SSE3, SSE4.x
  - On Linux see `/proc/cpuinfo` to identify supported SSE extensions

SSE ISA: Floating Point Instructions

- Floating point instructions operate on
  - Single (S) or double (D) precision operands
  - Scalar (S) or packed (P) operands
- Scalar operations update only lowest part of destination register
  - `ADDSD`: 

- Packed operations update all parts
  - `ADDPD`: 

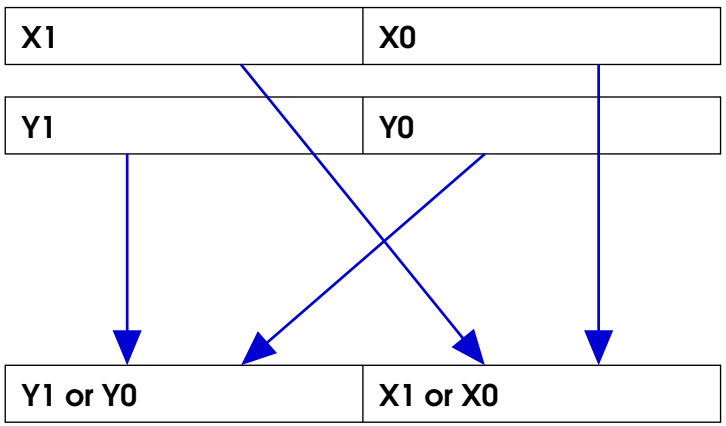
  - `ADDPS`:



- Horizontal add and subtract (SSE3)
  - `HADDPD`: 

SSE ISA: Shuffle Instructions

- `SHUFPS|SHUFPD imm8, xmm1, xmm2`:
  Merge two packed single/double precision floating-point values
- Example `SHUFPD`:



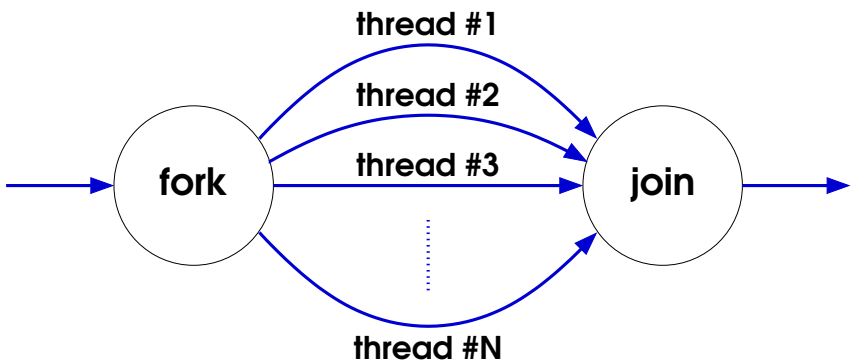
  Select operand: bit 0 (1) selects whether low or high part from destination (source) operand is used
- Use case: Swap X1 and X0 representing real and imaginary part of double precision complex number

SSE ISA: Example

```
1  #include <stdio.h>
2  #include <xmmintrin.h>
3
4  typedef union { __m128d v; double d[2]; } Pdouble;
5
6  int main(void) {
7    Pdouble a, b, c;
8
9    a.d[0] = 1.2; a.d[1] = 3.6;
10   b.d[0] = 0.8; b.d[1] = 2.4;
11
12   c.v = _mm_add_pd(a.v, b.v);
13
14   printf("c = (%.2f,%.2f)\n", c.d[1], c.d[0]);
15
16   return 0;
17 }
```

Thread Level Parallelism

- **Thread** = Smallest unit of processing that can be scheduled by operating system
- Single core use case: Hide pipeline stalls
  - Time-division multiplexing: If stall occurs then schedule other thread
- Typical execution model:



- Popular Application Programming Interfaces (API):
  - POSIX Threads
  - OpenMP

Simultaneous Multithreading (SMT)

- **SMT** architecture = Architecture which allows $>$ 1 thread per core to be executed simultaneously
  ☞ TLP and ILP are exploited simultaneously
- Architectural requirements
  - Dynamic management of resources, e.g.
    - Dynamic scheduling
    - Register renaming
  - Duplication of resources for each thread
    - Enlarged register file
    - Seperate program counters (PC)
  - Capability for instructions from multiple threads to commit
- Operating systems view: Multiple logical processors
- Also known as: Hyperthreading, hardware threading

Control Flow

- **Control flow** refers to the ability to change the order in which instructions are executed
  - Control flow instructions = **jump** or **branch** instructions
- Example loop

```
for (i = 0; i < 10; i++) {
  ... /* loop body */
}
```

  - At end of loop jump to beginning of loop
  - If condition `i < 10` is false, jump out of loop
- Types of control flow change
  - Conditional branches
  - Jumps
  - Procedure calls
  - Procedure returns

Control Flow (2)

- Control flow instructions must have a destination address
  - Run-time address (e.g. return address)
  - Compile time address ☞ **label**
- Jump instruction in x86 ISA: `jmp`
  Example for an (infinite) loop:

```
        instrA
.L2:
        instrB
        ...
        jmp .L2
```

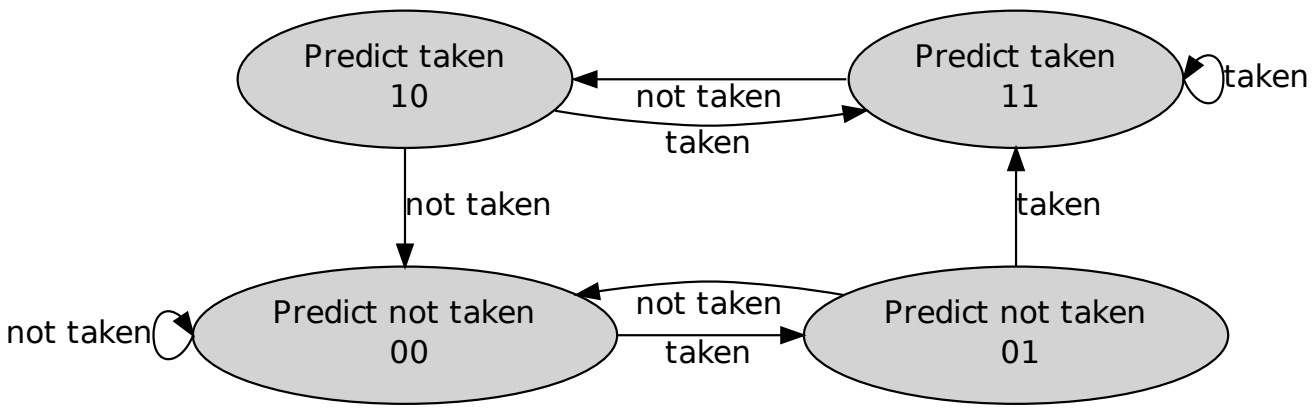
- Branching typically causes performance degradation
  - Pipelines must be drained and possibly re-filled
  - New instructions have to loaded

Branch Prediction

- Techniques to minimize costs of branching
  - Avoid branching
    - Code re-organisation
    - Use conditioned instructions (e.g., x86 ISA: `cmov`)
  - Loop unrolling
  - Branch prediction
- **Branch prediction** = Anticipate one branch being taken with higher probability then the other branch
  - Practical observation: individual branch is often highly biased towards taken or untaken
  - Static branch prediction
    - Compiler generates code to assume particular branch to be taken (e.g. branch hints instructions)
    - Decision based on “educated guess” or profiling information
  - Dynamic branch prediction

Branch Prediction Buffer

- 1-bit prediction scheme
  - Instantiate memory buffer with *n* bits
  - Depending on branch address select a bit
  - If bit is ’1’ take branch otherwise not
  - Flip bit in case of misprediction
- 2-bit prediction scheme: change from “predict taken” to “predict not taken” only after 2 mispredictions



Basic Block

- **Basic block** = Sequence of instructions with
  no branches in except to the entry
  and no branches out except at the exit
- Examples:
  - Loop body without control flow instructions, e.g.

```
for (i = 0; i < n; i++) {
  y[i] = x[i];
}
```

  - Subroutine without control flow instructions, e.g.

```
double pow2(double x) {
  double y;
  y = x * x;
  return y;
}
```

x86 ISA: Conditioned Instructions

- **Conditioned instruction** = Execution of conditioned instructions depends on (run-time) flags
- Conditioned instructions in x86 ISA:
  - Conditioned branch operations, e.g.

Jump if equal	`je`
Jump if greater	`jg`
Jump if less	`jl`
Jump if less or equal	`jle`

  - Conditional move `cmov`
    - Data transfer is only performed if condition is true
    - Use case: `k = (i == 0) ? 0 : k`

Assembler Generation

- Example GNU C-compiler: `gcc -S -O0 myprog.c`
  - Command will generate output file `myprog.s`
  - Usually better to start with zero-level optimization (`-O0`)
- Use cases:
  - Verification of sequence of instructions generated by compiler
  - Starting point for own assembler implementations

Assembler Generation Example

C-programm

```
void func()
{
  int i;
  for (i=0; i<100; i++);
}
```

Assembler:

```
        .file   "func.c"
        .text
.globl func
        .type   func, @function
func:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        movl    $0, -4(%ebp)
        jmp     .L2
.L3:
        addl    $1, -4(%ebp)
.L2:
        cmpl    $99, -4(%ebp)
        jle     .L3
        leave
        ret
```