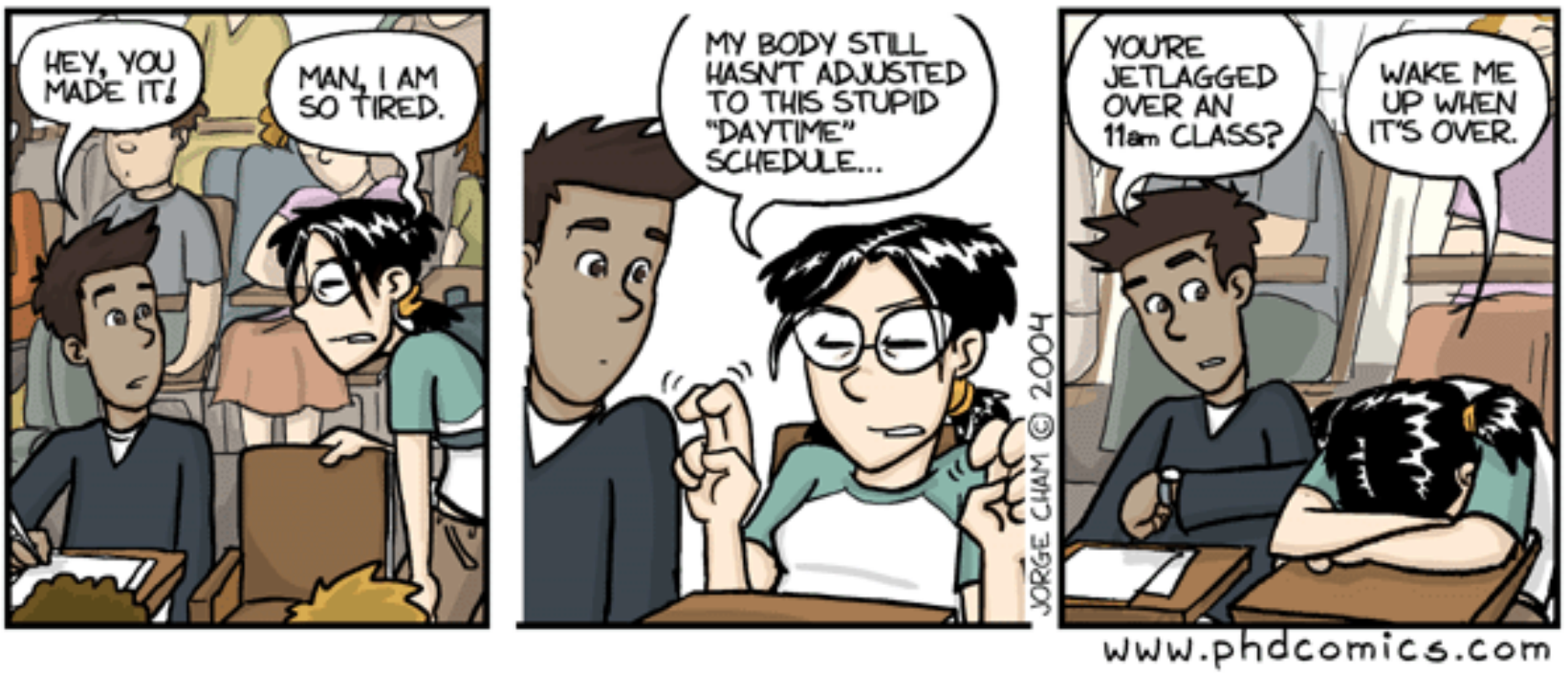
Introduction

L25: Modern Compiler Design



Course Aims

- Understand the performance characteristics of modern processors
- Be familiar with strategies for optimising dynamic dispatch for languages like JavaScript and Objective-C
- Have experience with algorithms for automatically taking advantage of SIMD, SIMT, and MIMD parallelism

Course Structure

- 8 Lectures
- 8 Supervised practical sessions
- Hands-on work with the LLVM compiler infrastructure

Assessment

- 4 short exercises
  - Simple pass / fail
  - Due: End of this term
- Longer assessed mini-project report
  - Up to 4,000 words
  - Due: Start of next term

LLVM

- Began as Chris Lattner's Masters' project in UIUC in 2002, supervised by Vikram Adve
- Now used in many compilers
  - ARM / AMD / Intel / nVidia GPU shader compilers
  - C/C++ compilers for various platforms
  - Lots of domain-specific languages
- LLVM is written in C++. This course will not teach you C++!

Questions?

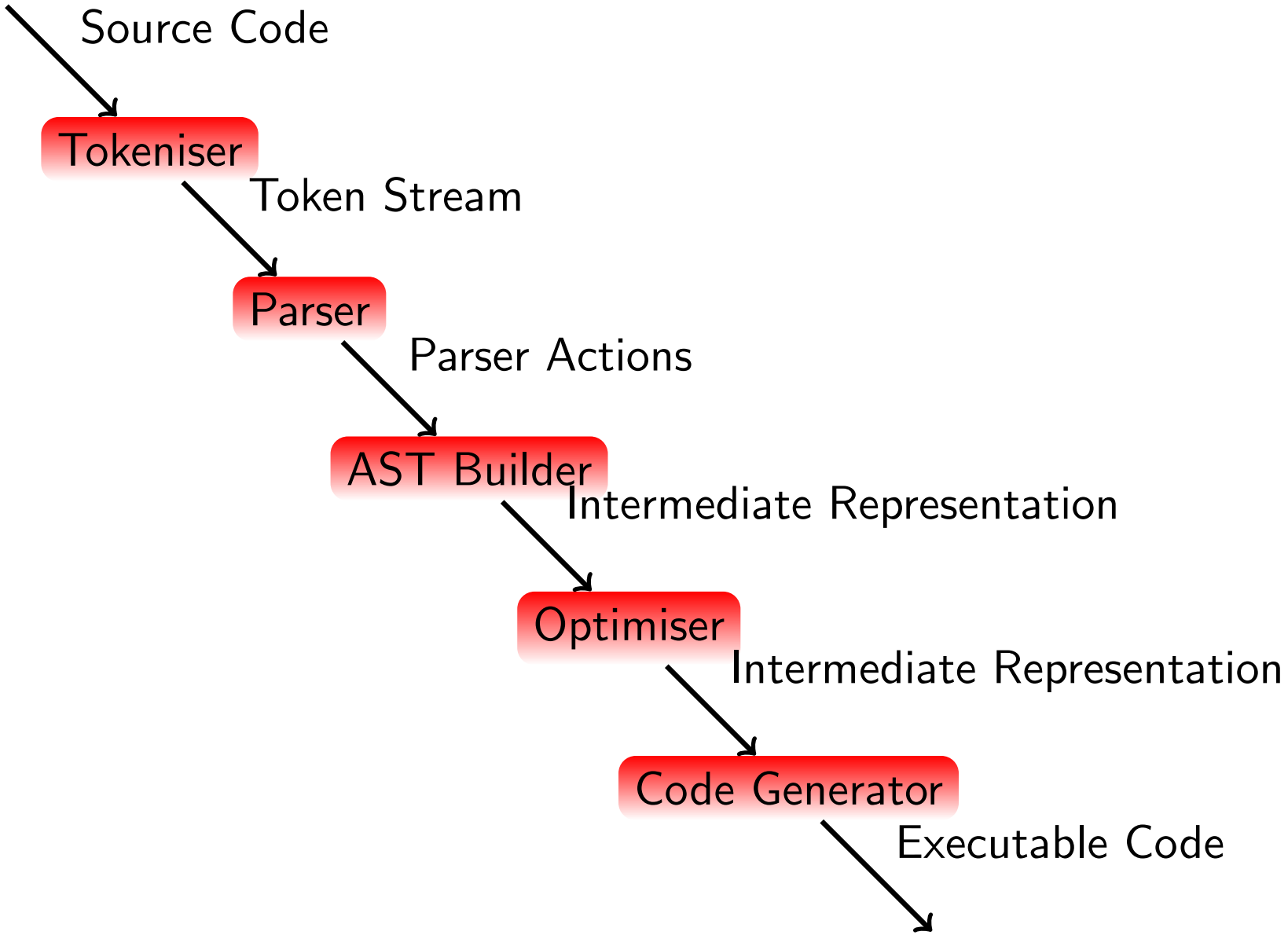
Modern Intermediate Representations (IR)

L25: Modern Compiler Design

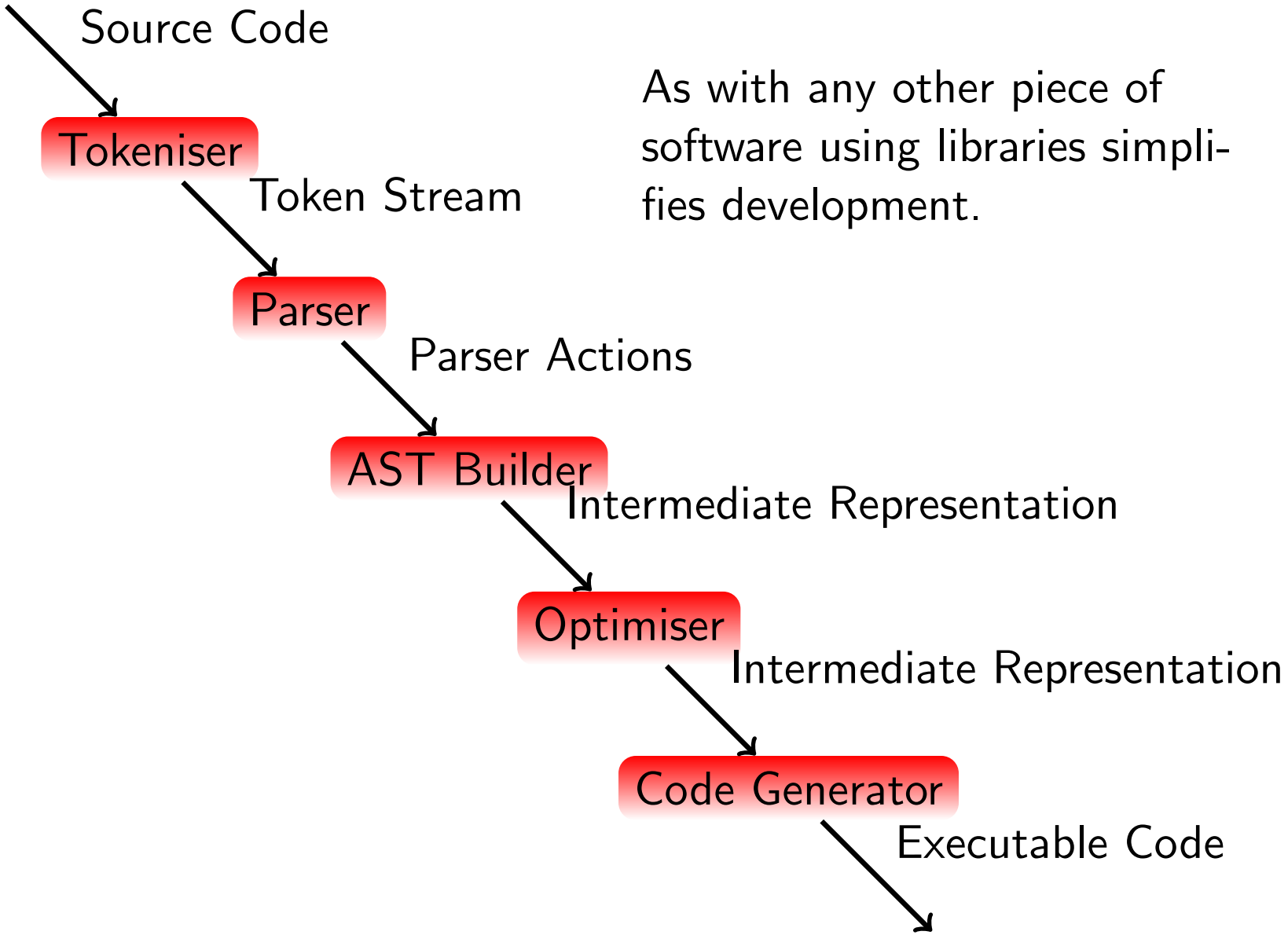
Reusable IR

- Modern compilers are made from loosely coupled components
- Front ends produce IR
- Middle ‘ends’ transform IR (optimisations)
- Back ends generate native code

Structure of a Modern Compiler



Structure of a Modern Compiler



As with any other piece of software using libraries simplifies development.

Optimisation Passes

- Modular, transform IR (Analysis passes just inspect IR)
- Can be run multiple times, in different orders
- May not always produce improvements in the wrong order!
- Some intentionally pessimise code to make later passes work better

Register vs Stack IR

- Stack makes interpreting, naive compilation easier
- Register makes various optimisations easier
- Which ones?

Common Subexpression Elimination: Register IR

Source language:

```
a = (b+c) * (b+c);
```

```
r1 = load b
r2 = load c
r3 = r1 + r2
r4 = load b
r5 = load c
r6 = r4 + r5
r7 = r3 * r6
store a r6
```

Common Subexpression Elimination: Register IR

Source language:

```
a = (b+c) * (b+c);
```

```
r1 = load b
r2 = load c
r3 = r1 + r2
~~r4 = load b~~
r5 = load c
r6 = r1 + r5
r7 = r3 * r6
store a r7
```

Common Subexpression Elimination: Register IR

Source language:

```
a = (b+c) * (b+c);
```

```
r1 = load b
r2 = load c
r3 = r1 + r2
~~r4 = load b~~
~~r5 = load c~~
r6 = r1 + r2
r7 = r3 * r6
store a r7
```

Common Subexpression Elimination: Register IR

Source language:

```
a = (b+c) * (b+c);
```

```
r1 = load b
r2 = load c
r3 = r1 + r2
~~r4 = load b~~
~~r5 = load c~~
~~r6 = r1 + r2~~
r7 = r3 * r3
store a r7
```

Common Subexpression Elimination: Stack IR

Source language:

```
a = (b+c) * (b+c);
```

```
load b
load c
add
load b
load c
add
mul
store a
```

Common Subexpression Elimination: Stack IR

Source language:

```
a = (b+c) * (b+c);
```

```
load b
load c
add
dup
mul
store a
```

Problems with CSE and Stack IR

- Entire operation must happen at once (no incremental algorithm)
- Finding identical subtrees is possible, reusing results is harder
- If the operations were not adjacent, must spill to temporary

Hierarchical vs Flat IR

- Source code is hierarchical (contains structured flow control, scoped values)
- Assembly is flat (all flow control is by jumps)
- Intermediate representations are supposed to be somewhere between the two

Hierarchical IR

- Easy to express high-level constructs
- Preserves program semantics
- Preserves high-level semantics (variable lifetime, exceptions) clearly
- Example: WHRIL in MIPSPro/Open64/Path64 and derivatives

Flat IR

- Easy to map to the back end
- Simple for optimisations to process
- Examples: LLVM IR, CGIR, PTX

Questions?

Modern Processor Architectures

L25: Modern Compiler Design

The 1960s - 1970s

- Instructions took multiple cycles
- Only one instruction in flight at once
- Optimisation meant minimising the number of instructions executed
- Sometimes replacing expensive general-purpose instructions with specialised sequences of cheaper ones

The 1980s

- CPUs became pipelined
- Optimisation meant minimising pipeline stalls
- Dependency ordering such that results were not needed in the next instruction
- Computed branches became very expensive when not correctly predicted

Stall Example


```
for (int i=100 ; i!=0 ; i--)
{
    ...
}
```

```
start:
    ...
    add r1, r1, 1
    jne r1, 0, start
```

Stall Example


```
for (int i=100 ; i!=0 ; i--)
{
    ...
}
```

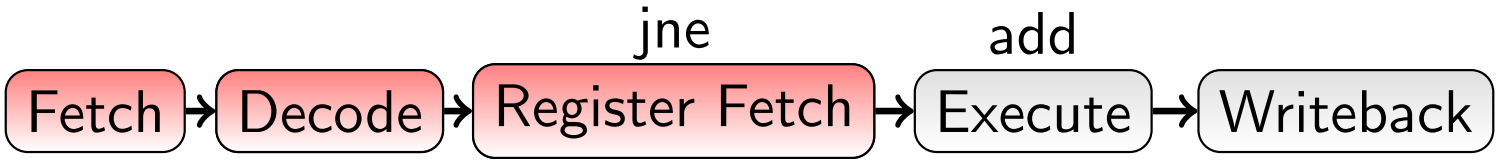
```
start:
    ...
    add r1, r1, 1
    jne r1, 0, start
```

Stall Example


```
for (int i=100 ; i!=0 ; i--)
{
    ...
}
```

```
start:
    ...
    add r1, r1, 1
    jne r1, 0, start
```

Stall Example




```
for (int i=100 ; i!=0 ; i--)
{
    ...
}
```

```
start:
    ...
    add r1, r1, 1
    jne r1, 0, start
```

Stall Example


```
for (int i=100 ; i!=0 ; i--)
{
    ...
}
```

```
start:
    ...
    add r1, r1, 1
    jne r1, 0, start
```

Stall Example


```
for (int i=100 ; i!=0 ; i--)
{
    ...
}
```

```
start:
    ...
    add r1, r1, 1
    jne r1, 0, start
```

Stall Example


```
for (int i=100 ; i!=0 ; i--)
{
    ...
}
```

```
start:
    ...
    add r1, r1, 1
    jne r1, 0, start
```

Fixing the Stall

```
for (int i=100 ; i!=0 ; i--)
{
    ...
}
```

```
start:
    add r1, r1, 1
    ...
    jne r1, 0, start
```

Fixing the Stall

```
for (int i=100 ; i!=0 ; i--)
{
    ...
}
```

```
start:
    add r1, r1, 1
    ...
    jne r1, 0, start
```

Is this a good solution?

The Early 1990s

- CPUs became much faster than memory
- Caches hid some latency
- Optimisation meant maximising locality of reference, prefetching
- Sometimes, recalculating results is faster than fetching from memory

The Mid 1990s

- CPUs became superscalar
  - Independent instructions executed in parallel
- CPUs became out-of-order
  - Reordered instructions to reduce dependencies
- Optimisation meant structuring code for highest-possible ILP
- Loop unrolling no longer such a big win

Superscalar CPU Pipeline Example: Sandy Bridge

Can dispatch up to six instructions at once, via 6 pipelines:

1. ALU, VecMul, Shuffle, FpDiv, FpMul, Blend
2. ALU, VecAdd, Shuffle, FpAdd
3. Load / Store address
4. Load / Store address
5. Load / Store data
6. ALU, Branch, Shuffle, VecLogic, Blend

Branch Predictors

- Achieve 95+% accuracy on modern CPUs
- No cost when branch is correctly predicted
- Long and wide pipelines mean very expensive for the remaining 5%!

The Late 1990s

- SIMD became mainstream
- Factor of 2-4$\times$ speedup when used correctly
- Optimisation meant ensuring data parallelism
- Loop unrolling starts winning again, as it exposes later optimisation opportunities (more on this later)

The Early 2000s

- (Homogeneous) Multicore became mainstream
- Power efficiency became important
- Parallelism provides both better throughput and lower power
- Optimisation meant exploiting fine-grained parallelism

The Late 2000s

- Programmable GPUs became mainstream
- Hardware optimised for stream processing in parallel
- Very fast for massively-parallel floating point operations
- Cost of moving data between CPU and CPU is high
- Optimisation meant offloading operations to the GPU

The 2010s

- Modern processors come with multiple CPU and GPU cores
- All cores behind the same memory interface, cost of moving data between them is low
- Increasingly contain specialised accelerators
- Often contain general-purpose (programmable) cores for specialised workload types (e.g. DSPs)
- Optimisation is hard.

Questions?