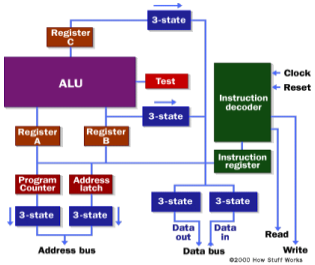
Учебный курс: Методы генерации тестовых программ для микропроцессоров

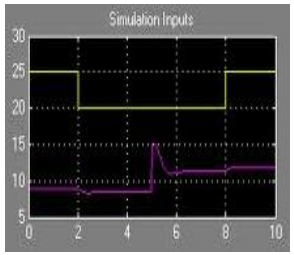
Александр Камкин, Андрей Татарников
{kamkin, andrewt}@ispras.ru

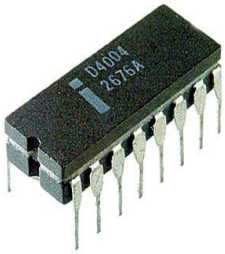
Institute for System Programming of the Russian Academy of Sciences (ISPRAS)
http://hardware.ispras.ru

Лекция 1. Введение: Проектирование микропроцессоров

Средства проектирования



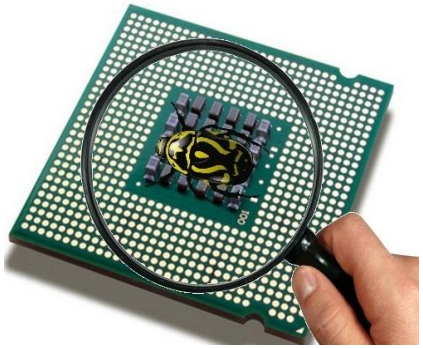




- Hardware Description Languages (HDL)
  - Verilog
  - VHDL
- Architecture Description Languages (ADL)
  - nML/Sim-nML
  - LISA
  - EXPRESSION

Лекция 2. Введение: Верификация микропроцессоров

Применяемые подходы

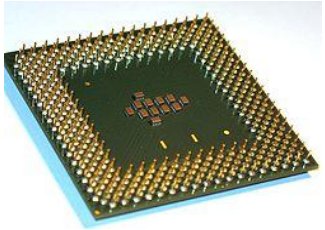


- Экспертиза
- Формальная верификация
- Имитационное тестирование
  - Модульное (сигналы)
  - Системное (тестовые программы)

Лекция 2. Введение: Верификация микропроцессоров

Системное тестирование

Тестовые программы



```
lui s1,0x27
ori s1,s1,0xc8
lui s3,0x4e
ori s3,s3,0xf7
...
```

Симулятор дизайна (VHDL, Verilog)

Эталонная модель (эмулятор) на C

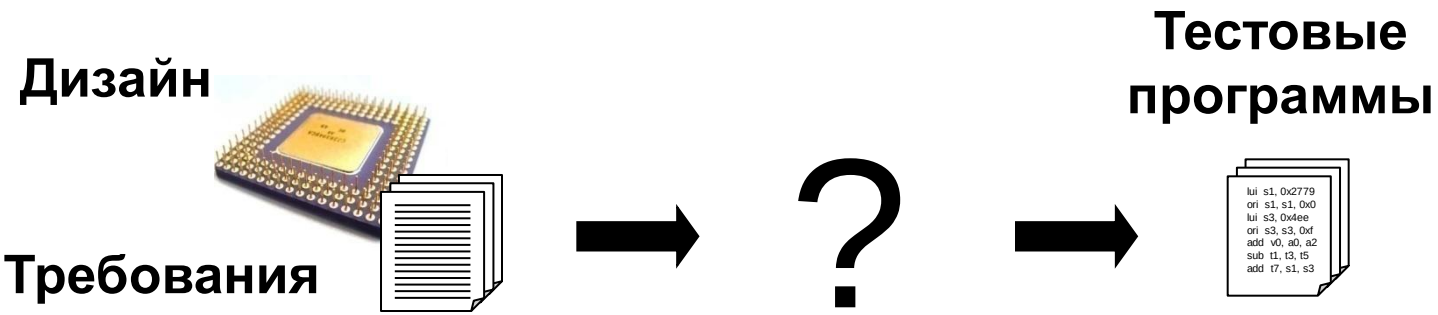
Сравнение трасс

✗ **Возникла ошибка**

☑ **Успешный прогон**

Лекция 2. Введение: Верификация микропроцессоров

Способы создания тестовых программ



- Ручная разработка
- Случайная генерация
- Генерация на основе тестовых шаблонов
- Генерация на основе моделей

Лекция 2. Введение: Верификация микропроцессоров

Подходы к генерации тестов

- Случайная
- Комбинаторная
- Основанная на ограничениях
- Направленная

Лекция 3. Введение: Основы MIPS

Общие сведения

- Разработан в Стэнфорде группой Джона Хеннесси (проект начат в 1981 году)
- Применяется в основном во встроенных системах
- Следует философии RISK
- Инструкции постоянной длины 32 бита
- Конвейер инструкций из 5 стадий
- Существуют 32 и 64-битные модификации (MIPS32 и MIPS64)

Лекция 3. Введение: Основы MIPS

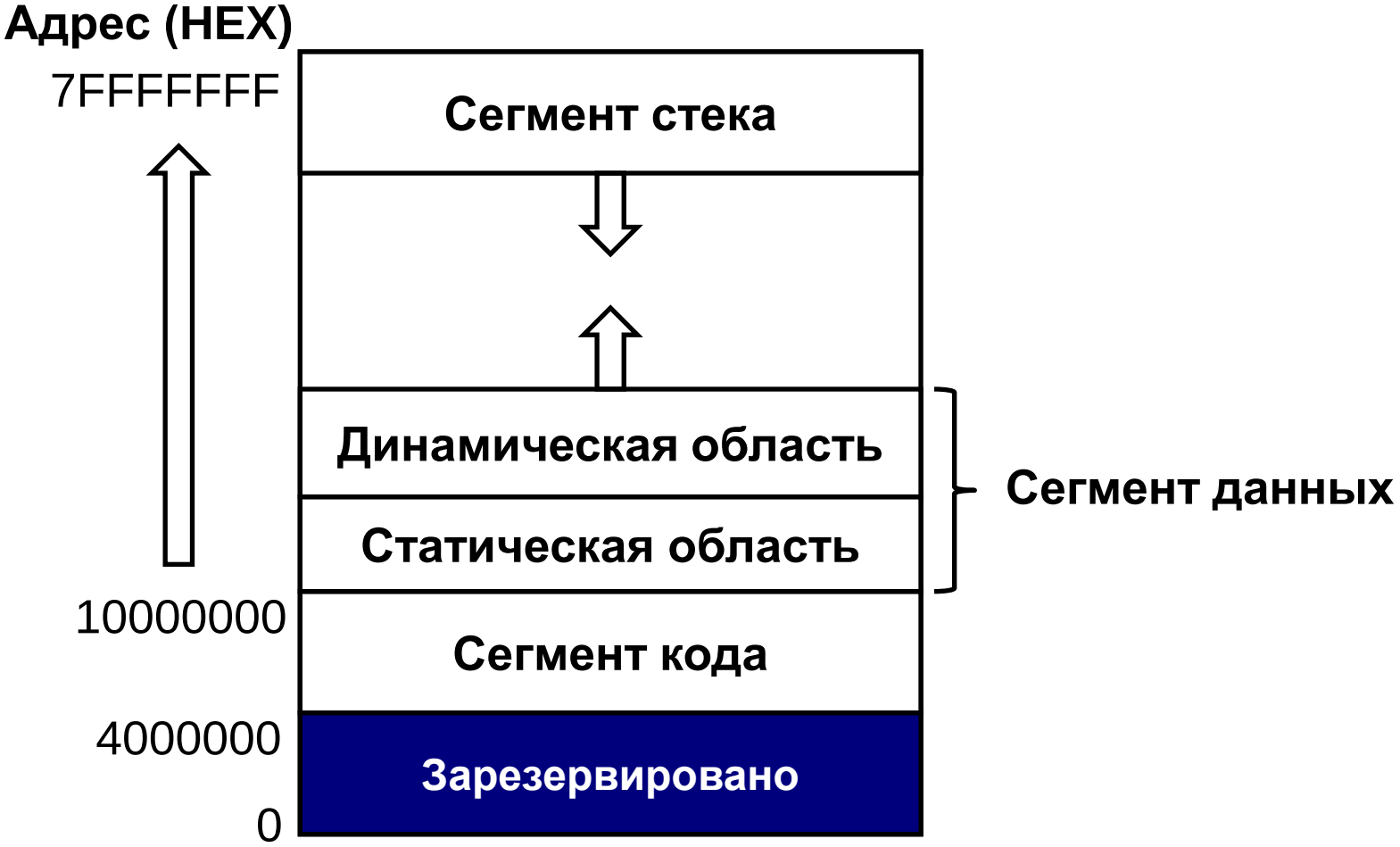
*Регистры MIPS**

- 32 регистра общего назначения (GPR)
- Счетчик команд (PC)
- Регистры умножения и деления (HI и LO)
- 32 регистра плавающей арифметики (FPR)
- 5 управляющих регистров плавающей арифметики (FCR{0, 25, 26, 28}, FCSR)

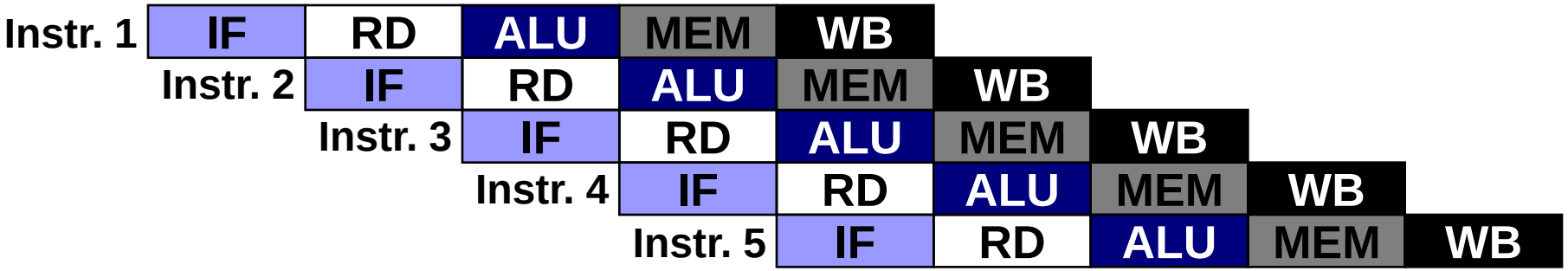
**** - Размер всех регистров 32 бита***

Структура памяти

Используемые сегменты: code, data, stack



Конвейер инструкций MIPS (5 стадий)



Стадия	Описание
IF (instruction fetch)	Читает инструкцию из памяти (кэш инструкций)
RD (read registers)	Извлекает данные из регистров
ALU	Выполняет операцию над данными
MEM	Читает/записывает данные в память (кэш данных)
WB (write back)	Сохраняет полученные данные в регистрах

Лекция 3. Введение: Основы MIPS

Регистры общего назначения

Имя	Номер	Назначение
$zero	0	Всегда хранит 0
$at	1	Зарезервирован для ассемблера
$v1, $v2	2,3	Возвращаемые значения функции
$a0-$a3	4-7	Аргументы функции (с 1-го по 4-й)
$t0-$t7	8-15	Временные, не сохраняются
$s0-$s7	16-23	Сохраняются после вызова функции
$t8, $t9	24, 25	Временные, не сохраняются
$k0, $k1	26, 27	Зарезервированы для ядра ОС
$gp	28	Указатель на область глобальных данных
$sp	29	Указатель стека
$fp	30	Указатель фрейма (если используется) или $s8
$ra	31	Адрес возврата

Инструкции MIPS

Типы инструкций

- Арифметические (Arithmetic)
- Логические (Logical)
- Передачи данных (Data transfer)
- Ветвления (Condition branch)
- Безусловного перехода (Unconditional jump)

Пример использования: A[12] = x + A[8]

lw $t0, 32($s3) *# $t0 := A[8] ($s3 хранит A)*

add $t0, $s2, $t0 *# $t0 := x + A[8] ($s2 хранит x)*

sw $t0, 48($s3) *# A[12] := x + A[8]*

Лекция 3. Введение: Основы MIPS

Формат инструкций MIPS

Register (R-Type)

31 – 26	25 – 21	20 – 16	15 – 10	11 – 6	5 – 0
opcode	rs	rt	rd	shamt	function
6	5	5	5	5	6

Immediate (I-Type)

31 – 26	25 – 21	20 – 16	15 – 0
opcode	rs	rt	imm
6	5	5	16

Jump (J-Type)

31 – 26	25 – 0
opcode	idx
6	26

opcode	Код операции	***shamt***	Сдвиг
rs	Регистр-получатель	***function***	Код функции
rt	Регистр-источник	***imm***	Константа
rd	Регистр-источник	***idx***	Индекс инструкции

Лекция 3. Введение: Основы MIPS
Основные инструкции (часть 1)

Имя	Ф-т	Тип	Пример	Значение
add	R	Arithmetic	add $s1, $s2, $s3	$s1 = $s2 + $s3
sub	R	Arithmetic	sub $s1, $s2, $s3	$s1 = $s2 - $s3
addi	I	Arithmetic	addi $s1, $s2, 20	$s1 = $s2 + 20
lw	I	Data transfer	lw $s1, 20($s2)	$s1 = Memory[$s2 + 20]
sw	I	Data transfer	sw $s1, 20($s2)	Memory[$s2 + 20] = $s1
and	R	Logical	and $s1, $s2, $s3	$s1 = $s2 & $s3
or	R	Logical	or $s1, $s2, $s3	$s1 = $s2 | $s3
nor	R	Logical	nor $s1, $s2, $s3	$s1 = ~($s2 | $s3)
andi	I	Logical	andi $s1, $s2, 20	$s1 = $s2 & 20
ori	I	Logical	ori $s1, $s2, 20	$s1 = $s2 | 20
sll	R	Logical	sll $s1, $s2, 10	$s1 = $s2 << 10
srl	R	Logical	srl $s1, $s2, 10	$s1 = $s2 >> 10

Лекция 3. Введение: Основы MIPS
Основные инструкции (часть 2)

Имя	Ф-т	Тип	Пример	Значение
ll	I	Data transfer	ll $s1, 20($s2)	$s1 = Memory*[$s2 + 20] 1st part of atomic swap
sc	I	Data transfer	sc $s1, 20($s2)	Memory [$s2 + 20] = $s1, $s1 = 0 or 1 2nd part of atomic swap
lui	I	Data transfer	lui $s1, 20	$s1 = 20 * 2^{16}
beq	I	Cond. branch	beq $s1, $s2, 25	if ($s1 == $s2) go to PC+4+100
bne	I	Cond. branch	bne $s1, $s2, 25	if ($s1 != $s2) go to PC+4+100
slt	R	Cond. branch	slt $s1, $s2, $s3	if ($s2 < $s3) $s1=1; else $s1=0
slti	I	Cond. branch	slti $s1, $s2, 20	if ($s2 < 20) $s1=1; else $s1=0
j	J	Uncond. jump	j 2500	go to 10000
jr	R	Uncond. jump	jr $ra	go to $ra
jal	J	Uncond. jump	jr 2500	$ra = PC + 4; go to 10000

Лекция 3. Введение: Основы MIPS

Разработка программ для MIPS

Симуляторы:

- **SPIM,** http://spimsimulator.sourceforge.net/
- **MARS,** http://courses.missouristate.edu/kenvollmar/mars/

Формат программы:

```
        .data
item:   .word 1          # Глобальные данные
        .text
        .globl main      # Объявляется глобальной
main:   lw $t0, item     # Пользовательский код
        …
```

Лекция 3. Введение: Основы MIPS

Компиляция программ на Си в код для MIPS

Язык Си:

```
void swap(int v[], int k)
{
  int temp;
  temp = v[k];
  v[k] = v[k+1];
  v[k+1] = temp;
}
```

Компилятор

Ассемблер

Ассемблер MIPS:

```
swap:
  sll $t1, $a1, 2
  add $t1, $a0, $t1
  lw $t0, 0($t1)
  lw $t2, 4($t1)
  sw $t2, 0($t1)
  sw $t0, 4($t1)
  jr $ra
```

Машинный код для MIPS:

```
00000000101000010000000000011000
00000000000110000001100000100001
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011111000000000000000001000
```

Лекция 4. Ручное тестирование: Арифметические операции

Общие сведения

- Ручная разработка
- Ручное создание тестовых данных
- Позитивные и негативные тесты
- Граничные условия
- Целочисленное переполнение
- Различные режимы адресации (регистры, константы и т.д.)

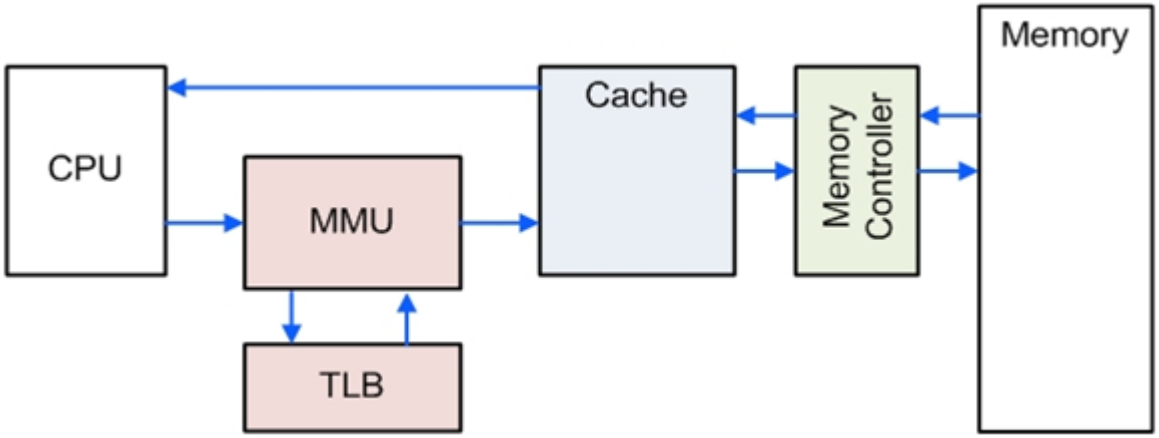
Лекция 4. Ручное тестирование: Арифметические операции

Простой тест для MIPS (SPIM)

```
  .data
success_msg:
  .asciiz "Success!"
failed_msg:
  .asciiz "Failed!"
  .text
  .globl main

main: addiu $s0, $0, 2
addiu $s1, $0, 3
addiu $s2, $0, 5

add $t0, $s0, $s1
bne $t0, $s2, fail

sub $t1, $s2, $s1
bne $t1, $s0, fail

la $a0, success_msg
li $v0,4
syscall

exit: li $v0,10
syscall

fail: la $a0, failed_msg
li $v0,4
syscall

j exit
```

Лекция 5. Ручное тестирование: Операции с памятью

Предметы тестирования



- Чтение/запись
- Трансляция адресов
- Попадания в кэш/промахи
- События виртуальной памяти

Лекция 5. Ручное тестирование: Операции с памятью

Тесты для MIPS

Различные комбинации инструкций lw и sw

Пример: работа со стеком

```
addi $sp,$sp,-12
sw $t1, 8($sp)
sw $t0, 4($sp)
sw $s0, 0($sp)

add $t0,$a0,$a1
add $t1,$a2,$a3
sub $s0,$t0,$t1
```

```
add $v0,$s0,$zero

lw $s0, 0($sp)
lw $t0, 4($sp)
lw $t1, 8($sp)
addi $sp,$sp,12

jr $ra
```

Ветвления в MIPS

Язык Си:

```
if (i == j)
f = g + h;
else f = g - h;
```

MIPS (i -> $s3, j -> $s4, f -> $s0, g -> $s1, h -> $s2):

```
bne $s3,$s4,Else        # go to Else if i not = j
add $s0,$s1,$s2         # f = g + h
j Exit                  # jump out of the if
Else: sub $s0,$s1,$s2   # f = g - h
Exit:
```

Лекция 7. Автоматизация: Скриптовые генераторы

Общие сведения

- Создаются на скриптовых языках (Ruby, Perl, Python, Bash)
- Случайные тесты
- Комбинаторные тесты
- Нацеленные тесты (граничные условия, специфические ситуации)
- Тестовые примеры описываются вручную
- Тестовые данные явно задаются

Лекция 7. Автоматизация: Скриптовые генераторы

Недостатки подхода

- Трудоемкость
- Недостаточная гибкость
- Не может обеспечить необходимый уровень покрытия
- Требует навыков программирования
- Ограниченные возможности для повторного использования кода

Лекция 7. Автоматизация: Скриптовые генераторы

Простейший генератор (Ruby)

```
INSTR_R = ["add", "sub", "or", "nor"]
RAND      = Random.new(123)

def self.t_rand
  return RAND.rand(8..15) end

def self.s_rand
  return RAND.rand(16..23) end

def self.ir_rand
  return INSTR_R[RAND.rand(0..INSTR_R.size-1)] end

def self.run
  (1..20).each do |i|
     puts "%s $%d, $%d, $%d" % [ir_rand, t_rand, s_rand, s_rand] end
end

run
```

Лекция 8. Автоматизация: Генерация на основе спецификаций

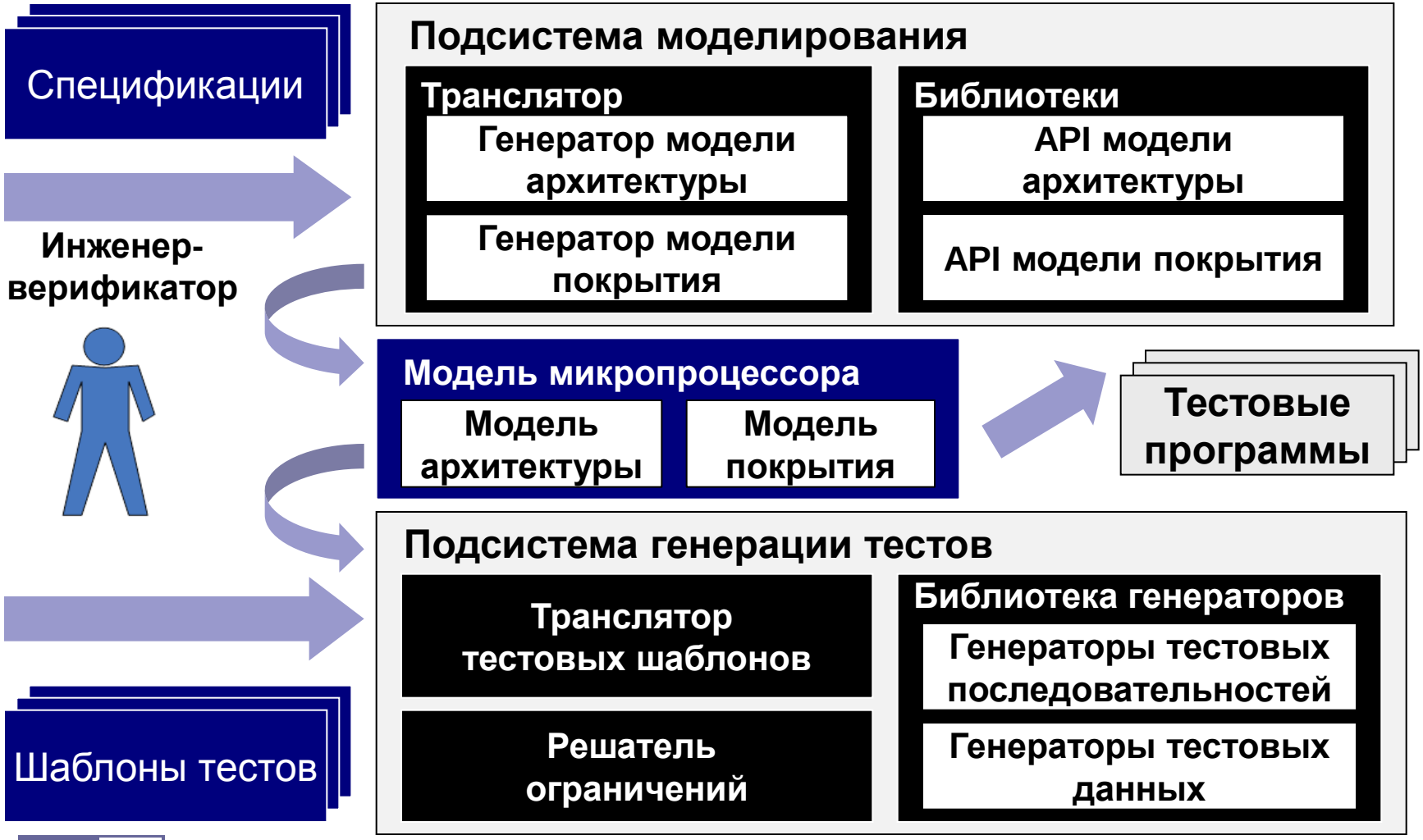
Описание подхода

- Использование предметно-ориентированных языков для настройки генератора
- Автоматическое построение модели архитектуры
- Автоматическое извлечение модели покрытия
- Возможность автоматического создания тестовых примеров
- Быстрая адаптация к изменениям в архитектуре
- Повторное использование логики ядра генератора

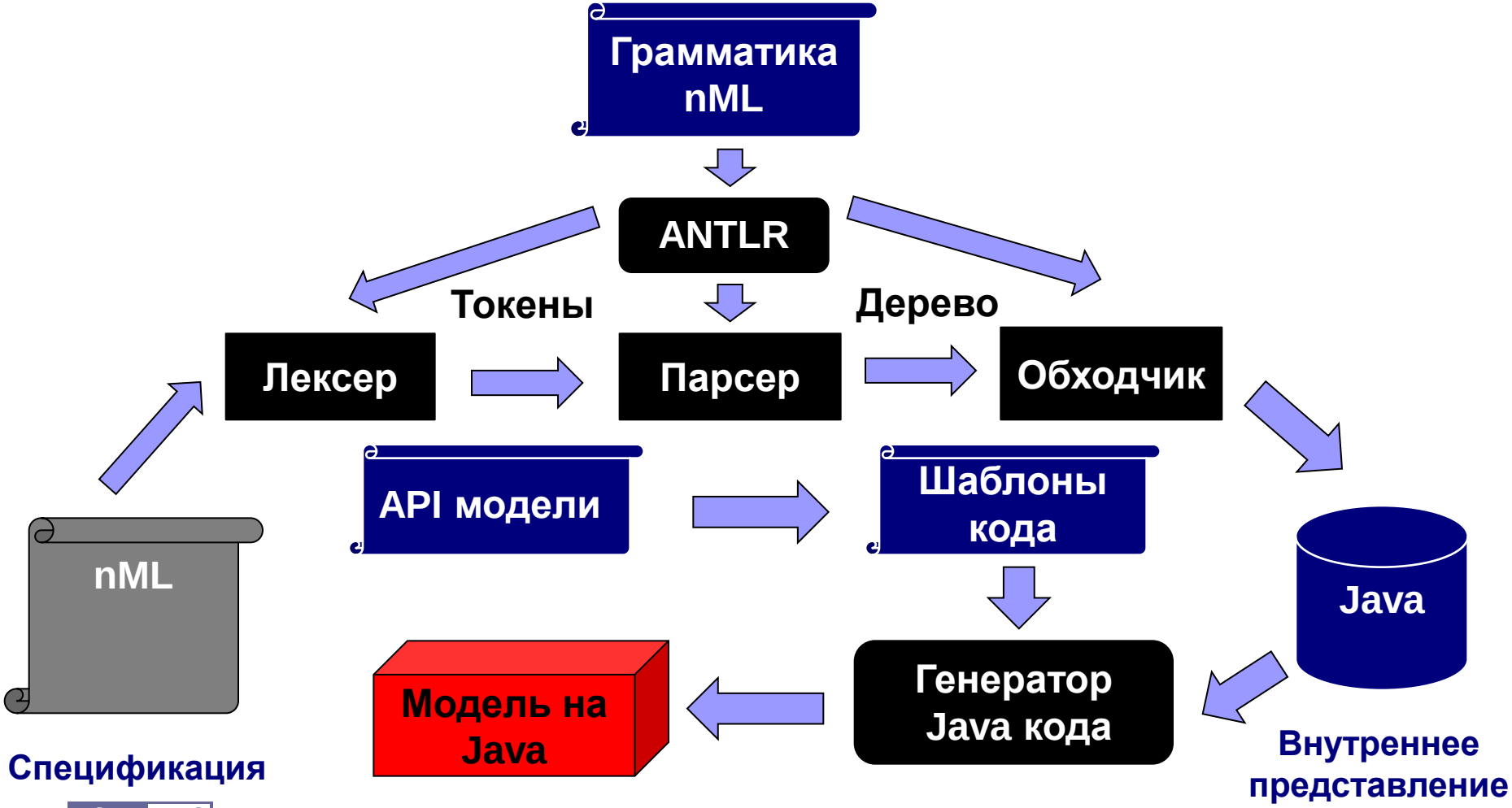
Лекция 8. Автоматизация:

Генерация на основе спецификаций

Схема генератора MicroTESK

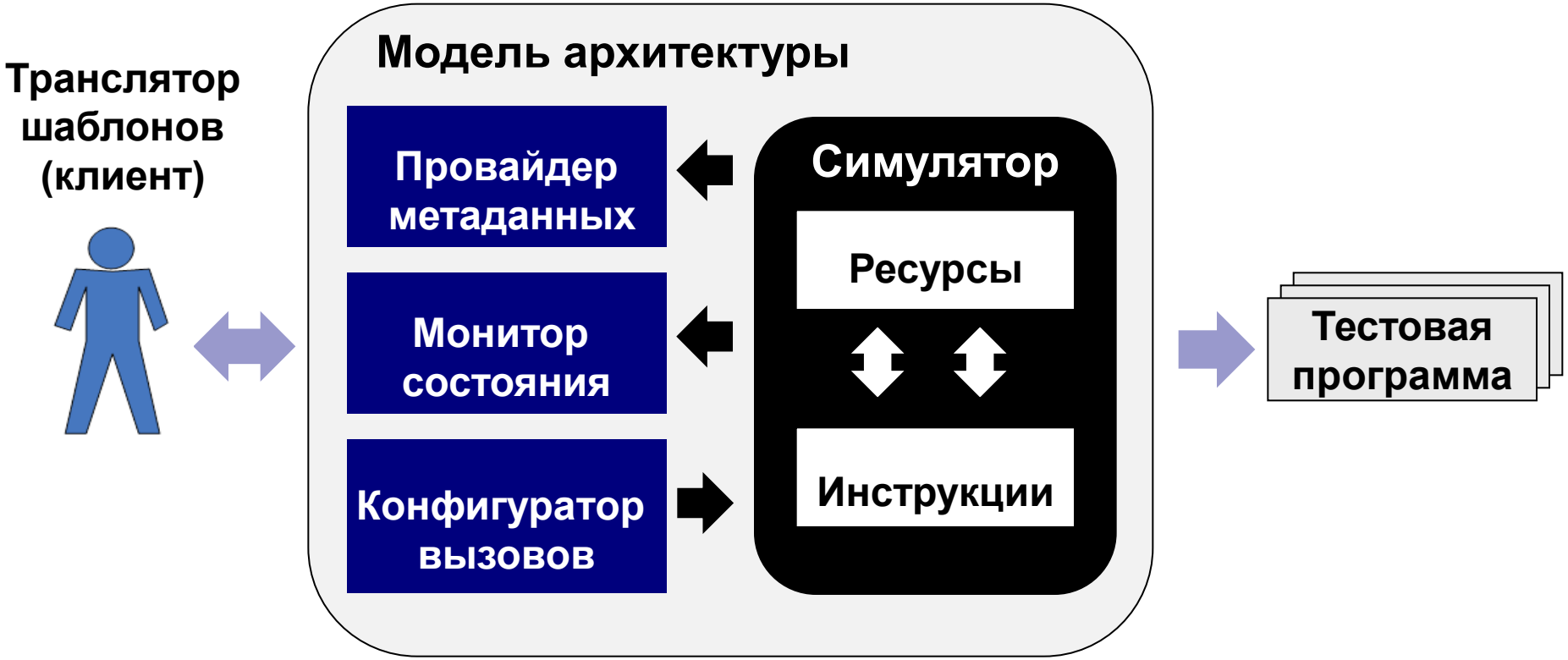


Лекция 8. Автоматизация:
Генерация на основе спецификаций
Схема трансляция спецификации



Лекция 8. Автоматизация: Генерация на основе спецификаций

Концептуальная схема модели



Лекция 8. Автоматизация: Генерация на основе спецификаций

Извлечение информации о покрытии

```
op ADD(rd: GPR, rs: GPR, rt: GPR)
action = {
   if(NotWordValue(rs) || NotWordValue(rt)) then
      UNPREDICTABLE();
   endif;
   tmp_word = rs<31..31>::rs<31..0> + rs<31..31>::rt<31..0>;
   if(tmp_word<32..32> != tmp_word<31..31>) then
      SignalException("IntegerOverflow");
   else
      rd = sign_extend(tmp_word<31..0>);
   endif;
}

syntax = format("add %s, %s, %s",  rd.syntax, rs.syntax, rt.syntax)

op ALU = ADD | SUB | ...
```

Предусловие

Тестовые ситуации

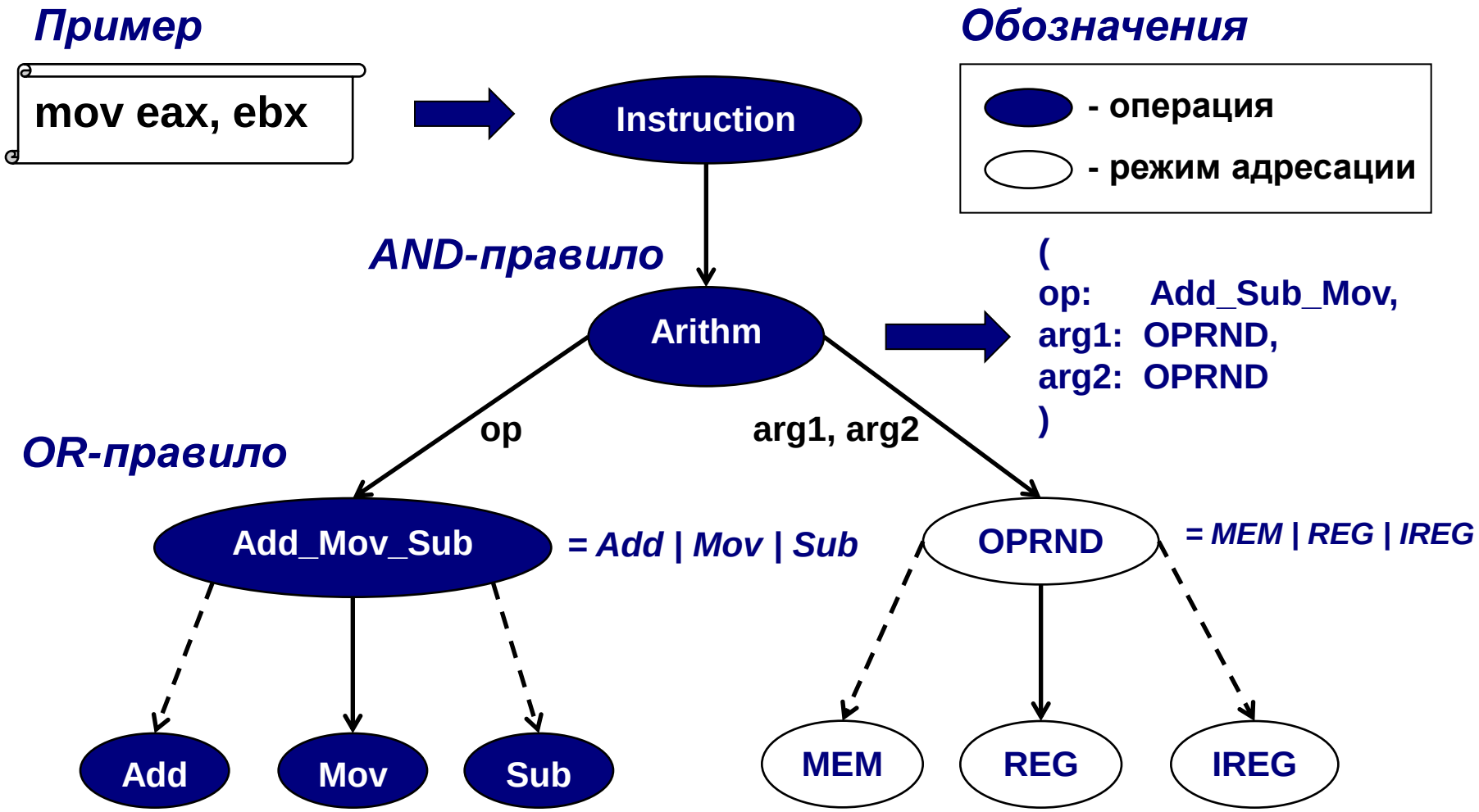
Классы эквивалентности

Общие сведения

- Язык описания архитектуры (Architecture Description Language, сокр. ADL)
- Разработан в TU Berlin в начале 90’х
- Описывает систему команд микропроцессора как иерархическую структуру
- Основан на атрибутной грамматике
- Применяется для создания симуляторов и генераторов кода

Лекция 9. Спецификация: Введение в nML

Структура формализма



Спецификация ресурсов микропроцессора

Константы и метки:

```
let  MSIZE = 2 ** 6
let  REGS  = 16
let  byte_order = ”little”
let  SP = ”GPR[13]”
```

Типы данных:

```
type  index  = card(6)
type  nibble = card(4)
type  byte_t = int(8)
```

Регистры:

```
reg  R[REGS, byte_t]
reg  PC[1, byte_t]
```

Память:

```
mem  M[MSIZE, byte_t]
```

Переменные:

```
var  temp[1, byte_t]
var  shifter_carry_out[1, card(1)]
var  ALU_OUT[1, card(32)]
```

Спецификация команд
Операции и режимы адресации

Режимы (правила AND и OR):

```
mode REG(i: nibble) = R[i]
syntax = format("R%d", i)
image  = format("01%4b", i)

mode OPRND =
        MEM | REG | IREG
```

Операции (правила AND и OR):

```
op Add(dest: OPRND, src: OPRND)
syntax = "add"
image  = "00"
action  = { dest = dest + src; }

op Add_Sub_Mov =
        Add | Sub | Mov
```

Корневая операция (точка входа):

```
op instruction(child: Add_Sub_Mov)
syntax = child.syntax
image  = child.image
action  = {
    child.action;
    PC = PC + 2;
}
```

Лекция 10. Спецификация: Описание операций микропроцессора MIPS

Спецификация ресурсов

Константы:

```
let REGBITS = 5
```

Типы данных:

```
type      bit = card(1)
type   byte_t = card(8)
type halfword = card(16)
type     word = card(32)
type   long_t = int(32)
type  address = card(32)
type    index = card(REGBITS)
```

Память:

```
mem  M[2**31, byte_t]
```

Метки:

```
let byte_order = "big"
let PC = "NIA"
let SP = "GPR[29]"
```

Регистры:

```
reg  GPR [2 ** REGS, long_t]
reg   NIA [1, address]
reg    LO[1, long_t]
reg    HI [1, long_t]
```

Глобальные переменные:

```
mem  CIA [1, address]
var branch [1, bit]
```

Лекция 10. Спецификация: Описание операций микропроцессора MIPS

Спецификация режимов адресации

Регистры (для инструкций R):

```
mode REG(r : index) = GPR[r]
   syntax = format("$%d", r)
   image  = format("%5b", r)
```

→

Пример использования (nML):

op ADD (rd: REG, rs: REG, rt: REG)

Код Ruby шаблона:

add reg(0), reg(1), reg(2)

Константы (для инструкций I):

```
mode IMM16(n : int(16)) = n
   syntax = format("%d", n)
   image  = format("%16b", n)
```

→

Пример использования (nML):

op ADDI (rt: REG, rs: REG, i: IMM16)

Код Ruby шаблона:

addi reg(0), reg(1), imm16(20)

Константы (для инструкций J):

```
mode IMM26(n : int(26)) = n
   syntax = format("%d", n)
   image  = format("%26b", n)
```

→

Пример использования (nML):

op J (target : IMM26)

Код Ruby шаблона:

j imm26(0x1000000)

Лекция 10. Спецификация:

Описание операций микропроцессора MIPS

Спецификация операций (часть 1)

Точка входа:

```
op instruction (child: instr_kind)
   syntax = child.syntax
   image  = child.image
   action = {
      CIA = NIA;
      if  branch == 0 then
         NIA = CIA + 4;
      else
         NIA = JMPADDR;
         branch = 0;
      endif;
      child.action;
      GPR[0] = 0;
   }
```

Группы операций (структура):

```
op instr_kind  = alu_instr
               | load_store_instr
               | branch_instr
               | jump_instr
op alu_instr   = arith_instr
               | logic_instr
op arith_instr = ADD
               | SUB
               | ADDI
               ...
op logic_instr = AND
               | OR
               | NOR
               ...
```

Лекция 10. Спецификация:

Описание операций микропроцессора MIPS

Спецификация операций (часть 2)

Операция сложения (ADD) :

```
var tmp_unsigned_word [1, card(32)]  // Временная переменная

op ADD (rd : REG, rs : REG, rt : REG)
   syntax = format ("ADD %s, %s, %s", rd.syntax, rs.syntax, rt.syntax)
   image  = format ("000000%s%s%s00000100000",
                    rs.image, rt.image, rd.image)
   action  =  {
      overflow_bit::tmp_unsigned_word = rs + rt;
      if overflow_bit == 1 then
         SignalException("Integer Overflow Exception");
      else
         rd = tmp_unsigned_word;
      endif;
   }
```

Лекция 10. Спецификация:

Описание операций микропроцессора MIPS

Спецификация операций (часть 3)

Операция условного перехода (BEQ):

```
var tmp_signed_word [1, int(32)]  // Временная переменная
var JMPADDR [1, address]          // Временная переменная

op BEQ (rs : REG, rt : REG, offset : IMM16)
   syntax = format ("BEQ %s, %s, %s", rs.syntax, rt.syntax, offset.syntax )
   image = format ("000100%s%s%s", rs.image, rt.image, offset.image )
   action = {
      if  rs == rt  then
         branch = 1;
         tmp_signed_word = offset;
         tmp_signed_word = tmp_signed_word << 2;
         JMPADDR = NIA + tmp_signed_word;
      endif;
   }
```

Лекция 10. Спецификация:

Описание операций микропроцессора MIPS

Спецификация операций (часть 4)

Операция сравнения значения регистра и константы (SLTI):

```
op SLTI (rt : REG, rs : REG, imm : IMM16)
   syntax = format ("SLTI %d, %s, %s", rt.syntax, rs.syntax, imm.syntax)
   image  = format ("001010%s%s%s", rs.image, rt.image, imm.image)
   action  = {
      if  rs < imm then
         rt = 1;
      else
         rt = 0;
      endif;
   }
```

Лекция 10. Спецификация:

Описание операций микропроцессора MIPS

Спецификация операций (часть 5)

Операция безусловного перехода (J):

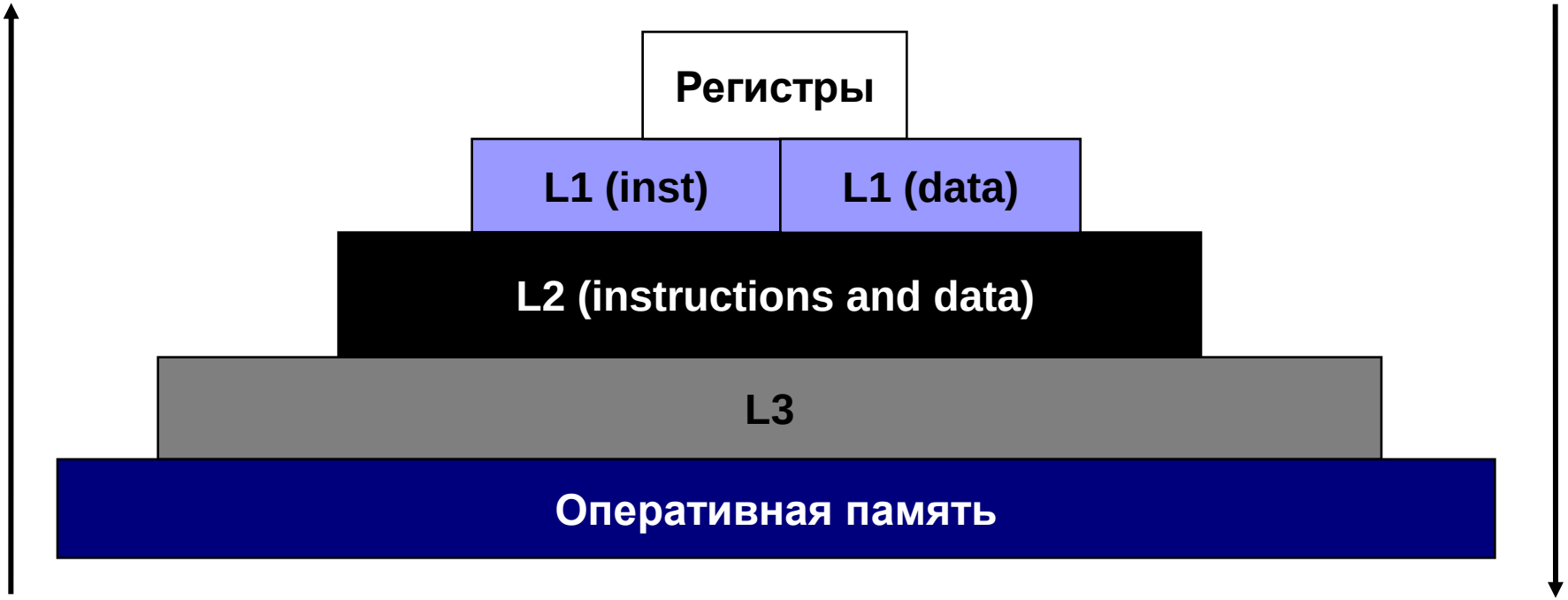
```
op J (target : IMM26)
   syntax = format ("J %s", target.syntax)
   image  = format ("000010%s", target.image)
   action  = {
      branch = 1;
      CIA = NIA;
      JMPADDR = CIA & 0xF0000000;
      JMPADDR = JMPADDR | ( target << 2);
   }
```

Лекция 11. Спецификация: Описание иерархии памяти

Схема иерархии памяти

Увеличение скорости чтения/записи
Уменьшение времени хранения

Увеличение ёмкости
Увеличение времени доступа



Лекция 11. Спецификация: Описание иерархии памяти

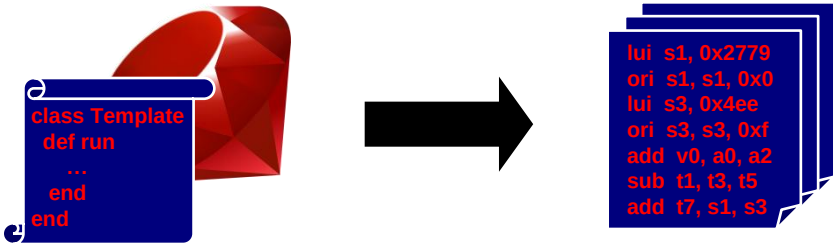
Спецификация буфера

```
buffer L1 {
        associativity = 4
        sets = 128
        line = (tag:30, data:256)
        index(addr:PA) = addr<9..8>
        match(addr:PA) = line.tag == addr<39..10>
        policy = LRU
    }
```

- Уровень ассоциативности
- Число множеств
- Структура блока данных (поля и их размер)
- Функция вычисления позиции данных на основе адреса
- Предикат проверки промаха/попадания
- Стратегии замещения данных при промахе

Лекция 12. Генерация: Разработка тестовых шаблонов

Генерация на основе шаблонов



- Абстрактное описание тестового примера
- Предусловия и постусловия
- Тестовые ситуации и ограничения
- Методы составления и комбинирования последовательностей инструкций
- Интеграция различных методов генерации
- Высокоуровневый язык описания (Ruby)
- Расширяемая среда

Лекция 12. Генерация: Разработка тестовых шаблонов

Схема генерации

Шаблон на Ruby:

```
block {
    eor r(0), r(0), register0
    add reg(2), reg(1), register0 do normal end
    mov_immediate reg(1), immediate(0, 0xFF)
    add reg(1), reg(4), register4 do random end
    sub reg(3), reg(2), register1 do overflow end
}
```

Генерация тестовых последовательностей

Генерация тестовых данных

Симуляция на модели

Ассемблерный код:

Инициализатор →

```
EOR R1, R1, R1
MOV R1, R1, LSL #8
ADD R1, R1, #df<<<0*2
MOV R1, R1, LSL #8
ADD R1, R1, #56<<<0*2
MOV R1, R1, LSL #8
ADD R1, R1, #bf<<<0*2
MOV R1, R1, LSL #8
ADD R1, R1, #ef<<<0*2
```

Тестовый код →

```
EOR R0, R0, R0
ADD R2, R1, R0
MOV R1, #ff<<<0*2
ADD R1, R4, R4
SUB R3, R2, R1
```

Лекция 12. Генерация:
Разработка тестовых шаблонов
Библиотека описания шаблонов

```
class TestCase01 < Template
  def pre # Выполняется перед основным кодом (инициализирующий код)
    addi reg(17), reg(0), imm16(5)
    addi reg(18), reg(0), imm16(10)
    addi reg(19), reg(0), imm16(7)
  end
  def run # Основной код тестового примера
    add reg(8), reg(17), reg(18)
    add reg(9), reg(19), reg(20)
    sub reg(16), reg(8), reg(9)
  end
  def post # Выполняется после основного кода
    sw reg(16), reg(21), imm16(128)
  end
end # class TestCase01
```

Лекция 12. Генерация: Разработка тестовых шаблонов

Использование тестовых ситуаций

- Ассоциируются с отдельными инструкциями
- Описывают условия перехода в определённое состояние / выполнение определённой ветви логики
- Возможно комбинирование тестовых ситуаций
- Возможно задавать вероятностное распределение

Пример:

```
class TestCase01 < Template
  def run
    add reg(8), reg(16), reg(17) do overflow([0.5]) end
    add reg(9), reg(18), reg(19) do normal end
  end
end
```

Вызывает целочисленное переполнение с вероятностью 50%

Гарантирует выполнение без переполнения

Лекция 12. Генерация:
Разработка тестовых шаблонов
Составление последовательностей (часть 1)
Блоки инструкций. Синтаксис.

```
class TestCase01 < Template
  def run
    block (:combine => “product”, :compose => “random”) {
      block (:engine => “random”, :length => 3, :count => 2) {  # Nested Block A
        add reg(8), reg(16), reg(17)
        sub reg(9), reg(18), reg(19)
        and reg(10), reg(16), reg(17)
        or reg(11), reg(18), reg(19)
      }
      block (:engine => "permutate") {  # Nested Block B
       lw reg(12), reg(20), imm16(0)
       sw reg(9), reg(21), imm16(0)
      }
    }
  end
end
```

Лекция 12. Генерация:
Разработка тестовых шаблонов
Составление последовательностей (часть 2)
Сгенерированный код

```
# Combination (1, 1)
sub $9, $18, $19 # Block A
lw $12, 0($20) # Block B
or $11, $18, $19 # Block A
sw $9, 0($21) # Block B
add $8, $16, $17 # Block A

# Combination (1, 2)
sw $9, 0($21) # Block B
sub $9, $18, $19 # Block A
lw $12, 0($20) # Block B
or $11, $18, $19 # Block A
add $8, $16, $17 # Block A
```

```
# Combination (2 , 1)
and $10, $16, $17 # Block A
and $10, $16, $17 # Block A
lw $12, 0($20) # Block B
add $8, $16, $17 # Block A
sw $9, 0($21) # Block B

# Combination (2, 2)
and $10, $16, $17 # Block A
sw $9, 0($21) # Block B
and $10, $16, $17 # Block A
lw $12, 0($20) # Block B
add $8, $16, $17 # Block A
```

Лекция 12. Генерация: Разработка тестовых шаблонов

Составление последовательностей (часть 3)

Стандартные генераторы последовательностей

Типы генераторов

- Комбинаторы (комбинирование инструкций)
- Композиторы (объединение последовательностей)

Стандартные комбинаторы

- Random
- Product
- Diagonal

Стандартные композиторы

- Random
- Catenation
- Nesting

Лекция 13. Генерация: Описание тестовых ситуаций

Тестовые ситуации

- Целевые состояния
- Целевые ветви логики
- Условия достижения/выполнения

Ограничения

- Предусловия
- Интервалы возможных значений
- Зависимости между входными данными
- Псевдослучайные значения

Лекция 13. Генерация: Описание тестовых ситуаций

Средства описания и решения ограничений

- Язык SMT-LIB, http://www.smtlib.org/
- Решатель Z3 (Microsoft Research), http://z3.codeplex.com/
- Решатель Yices, http://yices.csl.sri.com/
- Библиотека Java Constraint Solver API, http://forge.ispras.ru/projects/solver-api/

Лекция 13. Генерация: Описание тестовых ситуаций

Описание ограничений (часть 1)

Целочисленное переполнение на SMT-LIB

```
(define-sort   dword64() (_ BitVec 64))
(define-fun          zero() dword64 (_ bv0 64))
(define-fun  base_size() dword64 (_ bv32 64))
(define-fun sign_mask() dword64 (bvshl (bvnot zero) base_size))

(define-fun IsValidPos ((x!1 dword64)) Bool (= (bvand x!1 sign_mask) zero))
(define-fun IsValidNeg ((x!1 dword64)) Bool (= (bvand x!1 sign_mask) sign_mask))
(define-fun IsValidInt ((x!1 dword64)) Bool (or (IsValidPos x!1) (IsValidNeg x!1)))

(declare-const rs dword64)
(declare-const rt dword64)

(assert (and (IsValidInt rs) (IsValidInt rt) (not (= rs rt))))
(assert (not (IsValidInt (bvadd rs rt))))

(check-sat)
(get-value (rs rt))
```

Лекция 13. Генерация: Описание тестовых ситуаций

Описание ограничений (часть 2)

Использование Java Constraint Solver API

```
final IDataType BIT_VECTOR =
    DataFactory.createDataType(EDataType.BIT_VECTOR, 32);
final Constraint constraint = new Constraint();
constraint.setName("SampleConstraint1");
constraint.setSolverId(ESolverId.Z3_TEXT);
final Variable x = new Variable(constraint.addVariable("x", BIT_VECTOR));
final Syntax syntax = new Syntax();
constraint.setSyntax(syntax);
syntax.addFormula(new Formula(
    new Operation(
        EStandardOperation.EQ, x, new Value(BIT_VECTOR.valueOf("100", 10))
        )));
return constraint;
```

Лекция 13. Генерация: Описание тестовых ситуаций

Описание ограничений (часть 3)

Использование языка XML

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<Constraint version="1.0">
   <Name>SampleConstraint1</Name><Description/><Solver id="Z3_TEXT"/>
   <Signature>
      <Variable length="32" name="x" type="BIT_VECTOR" value=""/>
   </Signature>
   <Syntax>
      <Formula>
         <Expression>
            <Operation family="ru.ispras.solver.core.syntax.EStandardOperation" id="EQ"/>
            <VariableRef name="x"/>
            <Value length="32" type="BIT_VECTOR" value="00000064"/>
         </Expression>
      </Formula>
   </Syntax>
</Constraint>
```

Лекция 14. Генерация: Создание специализированных генераторов

Генераторы последовательностей

- Классы композиторов (наследуют класс Compositor)
- Классы комбинаторов (наследуют класс Combinator)
- Классы генераторов последовательностей (реализуют интерфейс Generator)

Генераторы данных

- Решатели ограничений
- Генераторы псевдослучайных значений
- Генераторы инициализирующих последовательностей

Лекция 15. Углублённый материал: Описание конвейеров

Конфигурация конвейера

```
pipeline() {
    IF(I);
    ID(I);
    EX(I);
    MEM(I);
    WR(I);
}
...
stage EX(I: Instruction) {
    switch(I.type) f
    case ALU: EX_ALU(I);
    case FPU: EX_FPU(I);
    ...
}}
```

```
stage FPU(I: Instruction) throws
    Inexact, Overflow, Underflow {
    delay(10..15);
}
```

Лекция 16. Углублённый материал: Тестирование конвейеров

Конвейерные конфликты

Конфликты по данным

```
ADD R, ?, ?
...
SUB ?, R, ?
```

Данные

Исключения

IF ID ME EX WR

IF ID ME EX WR

IF ID ME EX WR

Ветвления

Структурные конфликты

```
DIV.D ?, ?, ?
...
MUL.S ?, ?, ?
```