Michał Stochmiałek <misto@pld-linux.org>
Tomasz Poradowski <batonik@batonik.net>

Dokumentacja projektu
SmArc

Politechnika Wrocławska, Wrocław, (c) 2002-2003

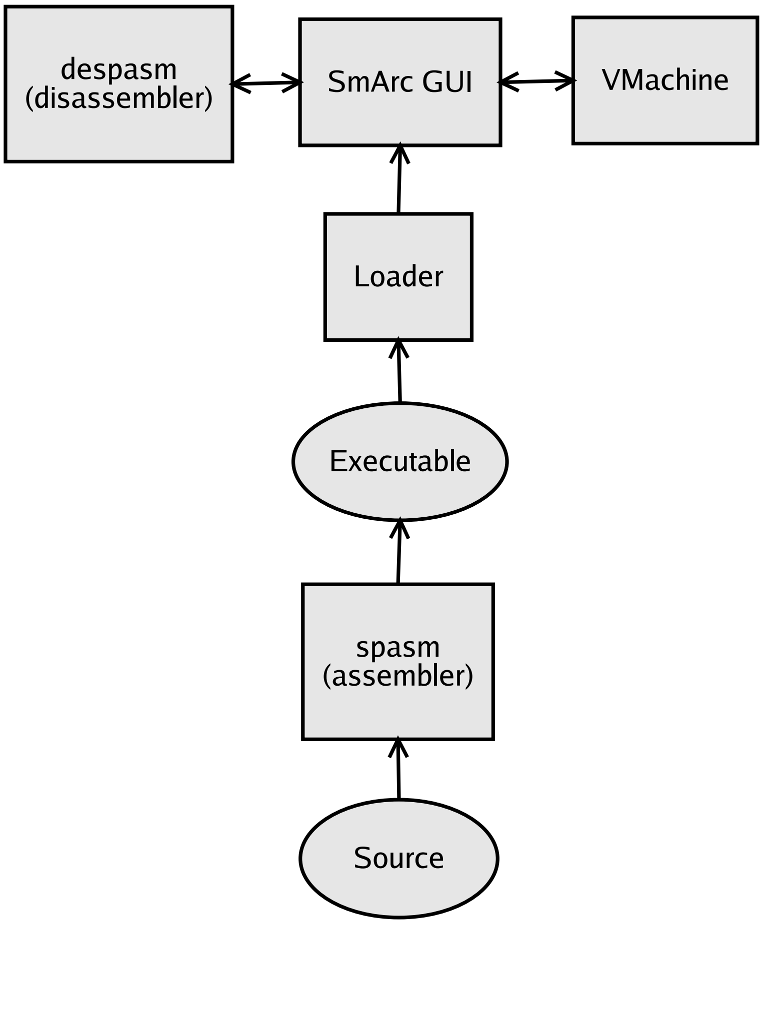
1. Wstęp

Wirtualna maszyna **SmArc** powstała jako projekt akademicki, mający na celu bliższe poznanie architektury oraz sposobu działania procesora SPARC. **SmArc** (od *Small SPARC*) zawiera jego wersję uproszczoną, jednakże wiernie oddaje ona realia i większość aspektów projektowych oraz technologicznych zastosowanych w oryginale (więcej informacji w specyfikacji SPARCv8[1]).

2. Elementy SmArc

W skład pakietu **SmArc** wchodzą następujące składniki (szczegółowy opis znajduje się w dalszej części):

- *vmachine* - wirtulana maszyna oparta na uproszczonej wersji procesora SPARC
- *spasm* - asembler programów napisanych w języku **SmArcAs** (uproszczonej wersji asemblera SPARC)
- *despasm* - deasembler plików wykonywalnych zapisanych w formacie maszyny wirtualnej **SmArc**
- *smarcgui* - interfejs graficzny wirtualnej maszyny **SmArc**
- loader - moduł służacy do wczytania plików wykonywalnych **SmArc**
- examples - przykładowe programy źródłowe w języku **SmArcAs**



2.1. vmachine

vmachine jest głównym modułem projektu **SmArc**. Jest to uproszczony model procesora SPARC o klasycznej strukturze potoku 5-stopniowego. Symuluje wykonywanie programów z dokładnością do mikrooperacji wykonywanych w jednym cyklu zegara. *vmachine* umożliwia zaobserwować przepływ danych w wewnętrznych rejestrach procesora, w szczególności sytuacje przestoju potoku (*stall*) oraz wyprzedzenia argumentów (*forwarding*).

Moduł *vmachine* nie posiada żadnego interfejsu użytkownika. Funkcje te są udostępniane przez moduł *smarcgui*, który jest pomostem pomiędzy użytkownikiem a symulatorem *vmachine*.

W rozdziale 3.1 uszczególnione zostały założenia, jakie zostały przyjęte podczas realizacji modułu *vmachine*.

2.2. spasm

Program *spasm* służy do tłumaczenia plików źródłowych napisanych w języku **SmArcAs** (uproszczonej wersji asemblera SPARC) do kodu wykonywalnego (binarnego) w formacie maszyny wirtualnej **SmArc**. *spasm* jest uruchamiany z linii poleceń:

```
spasm [-h] [-v] [-q] [-o <plikbin>] <plikasm>
```

gdzie:

`<plikasm>`	plik źródłowy zawierający program w języku **SmArcAs**.
`-h`	wyświetla krótką informację o składni wywołania programu
`-v`	włącza tryb *debug* - program będzie wtedy wyświetlał sporą ilość informacji dotyczących analizy składniowo-leksykalnej; nie zostaną także usunięte osobne pliki tymczasowe programu zawierające sekcje danych oraz kodu
`-q`	wyłącza informacje ostrzegawcze (o braku niektórych elementów składni języka, których brak nie powoduje jednakże błędów przy generowaniu pliku wynikowego) (użycie NIE ZALECANE)
`-o <plikbin>`	określa nazwę pliku wynikowego, do którego ma zostać zapisany program wykonywalny (!`<plikbin>`); jeżeli parametr ten zostanie pominięty, to zostanie użyta domyślna nazwa pliku wynikowego 'spasm.output'

Jeżeli podczas analizy pliku źródłowego nie wystąpiły żadne błędy, *spasm* nie wypisze żadnego komunikatu (zachowanie podobne do standardowych kompilatorów) oraz utworzy plik wykonywalny w formacie maszyny wirtualnej (opis poniżej); w przeciwnym przypadku na ekranie pojawią się komunikaty błędów i plik wynikowy nie zostanie utworzony (lub będzie pusty).

2.3. despasm

Program *despasm* to prosty deasembler kodu binarnego wygenerowanego za pomocą asemblera *spasm*. *despasm* jest domyślnie wywoływany bezpośrednio z interfejsu graficznego *smarcgui*. Jest jednak możliwe jego ręczne wywołanie:

```
despasm [-b] [-n] <prog>
```

gdzie:

`<prog>`	plik binarny w formacie wykonywalnym maszyny wirtualnej **SmArc**.
`-b`	powoduje, że pomijane jest wyświetlanie adresu danego rozkazu (liczonego od początku, czyli adresu 0x0) na początku każdej linii
`-n`	modyfikuje domyślny tryb wyświetlania nazw rejestrów w argumencie rozkazu z zapisu `%r<N>` do bardziej przyjaznego programiście zapisu alternatywnego - `%g<M>`, `%o<M>`, `%i<M>`, `%l<M>`, `%sp` oraz `%fp` (gdzie $0 \leqslant \mathtt{N} \leqslant 31$, $0 \leqslant \mathtt{M} \leqslant 7$).

Domyślnie wyświetla on dla każdego słowa rozkazowego na standardowym wyjściu linię postaci:

```
adres:   rozkaz   argumenty
```

2.4. smarcgui

smarcgui to graficzny interfejs maszyny wirtualnej **SmArc**. Intuicyjna realizacja zapewnia łatwy i zorganizowany dostęp do wewnętrznych elementów maszyny wirtualnej, takich jak rejestry i pamięć, oraz do stanów pośrednich w poszczególnych krokach przetwarzania potokowego.

2.5. loader

Moduł *loader* jest odpowiedzialny za wczytanie pliku wykonywalnego. Wykonuje on funkcje zredukowanego systemu operacyjnego:

- zidentyfikowanie oraz wczytanie pliku wykonywalnego
- przygotowanie sekcji danych oraz sekcji instrukcji do załadowania do pamięci *vmachine*
- przygotowanie małego kodu inicjalizującego
    - przesunięcie okna rejestrów
    - uruchomienie wczytanego programu
    - po powrocie programu (użycie komendy `ret`) przesunięcie okna rejestrów z powrotem
    - zatrzymanie wirtualnej maszyny (instrukcja o kodzie równym 0)

2.6. examples

W katalogu *examples* zawarte zostały poglądowe programy mające na celu pokazanie jednej lub więcej właściwości procesora SPARC. Niektóre z nich to zwykłe, typowe programy testowe.

- addloop.asm - pętla wykonywana zadaną ilość razy
- arith.asm - instrukcje arytmetyczne
- forward.asm - demonstracja *forwardingu* w procesorze SPARC
- load.asm - odczyt i zapis z/do pamięci DCache
- logic.asm - operacje logiczne na rejestrach
- loop.asm - instrukcje branch
- memcpy.asm - kopiowanie danych
- procedure.asm - instrukcja call wywołanie fragmentu programu (procedury)
- stall.asm - demonstracja wystąpienia *stall* w procesorze SPARC

3. Założenia i szczegóły techniczne

3.1. Wirtualna maszyna SmArc

vmachine jest implementacją klasycznej struktury potoku 5-stopniowego. Poszczególne stopnie potoku wykonują poniższe operacje.

- **IF** (*Instruction Fetch*) - pobranie rozkazu z pamięci rozkazów
- **ID** (*Instruction Decode*) - dekodowanie rozkazu, przygotowanie danych do kolejnej fazy: odczyt z rejestrów programowych, przygotowanie stałych, sprawdzenie warunków przy skokach (*branch*)
- **EX/AC** (*Execution/Address Calculation*) - wykonanie rozkazów arytmetycznych, logicznych, przesunieć, innych obliczeń arytmetycznych potrzebnych rozkazom sterującym, przygotowanie adresu w rozkazach typu *load* i *store*.
- **MA** (*Memory Access*) - zapis lub odczyt do/z pamięci danych
- **WB** (*Write Back*) - zapis wyniku w rejestrach programowych

Każdy stopień potoku jest wyraźnie wyodrębiony. Fazy komunikują się pomiędzy sobą za pomocą dwutaktowych (master-slave) rejestrów wewnętrzych. *vmachine* pozwala zaobserwować przepływ danych pomiędzy fazami udostępniając wartości poniższych rejestrów.

- **IF** (*Instruction Fetch*) - **nPC**
- **ID** (*Instruction Decode*) - **PC, IR**
- **EX/AC** (*Execution/Address Calculation*) - **pPC, A1, A2, R, DR**
- **MA** (*Memory Access*) - **T, pR, pDR**
- **WB** (*Write Back*) - **pT, D, ppDR**

3.1.1. Założenia SmArc i vmachine

Architektura **SmArc** w stosunku architektury SPARC została w celach dydaktycznych uproszczona. Zrezygnowano z implementacji poniższych:

- jednostka obliczeń zmiennoprzecinkowych, koprocesor
- przerwania, wyjątki i pułapki
- pewnej części listy instrukcji (patrz niżej)
- wewnętrzne rejestrów procesora **Y, WIM, TBR**, częściowo **PSR**

W stosunku do specyfikacji SPARCv8[1] do listy instrukcji architektury **SmArc** nie wchodzą:

- rozkazy związane z jednostką obliczeń zmiennoprzecinkowych oraz koprocesorem
- rozkazy związane z obsługa przerwań, pułapek, wyjątków
- *annul bit* w rozkazach warunkowych (*branch*)
- rozkazy związane z odczytem oraz zapisem z/do pamięci podwójnych słów *double-word* (64-bitowych)
- rozkazy mnożenia oraz dzielenia liczb całkowitych
- rozkazy związane z odczytem/zapisem niezaimplementowanych rejestrów (**Y, WIM, TBR**) lub częściowo zaimplementowanych **PSR**
- `ldstub, swap, stbar, flush`

vmachine implementuje wszystkie instrukcje, jakie generuje assembler *spasm* (opis w rozdziale 3.2). Instrukcje zrealizowane w wirtualnej maszynie **SmArc** są zgodne z opisem zawartym w specyfikacji SPARCv8 [1].
Inne założenia symulatora *vmachine*:

- 32 okien rejestrów, czyli w sumie 520
- rozdział na pamięć danych i instrukcji
- 256Kb pamięci danych
- 256Kb pamięci instrukcji
- słowo rozkazowe o kodzie o wartości 0 zatrzymuje symulator (po zakończeniu działania instrukcji znajdujących się już w potoku)

3.2. Język SmArcAs

Składnia języka **SmArcAs** (*Small SPARC Assembler*):

```
    program  ::= [linia]*
      linia  ::= [wyrażenie] [!komentarz]
  wyrażenie  ::= [etykieta:] [instrukcja]
 instrukcja  ::= rozkaz | pseudorozkaz | dyrektywa
```

gdzie:
`[ ]` - element opcjonalny
`*` - powtórzenie 0 lub więcej razy
`|` - alternatywa

Rozpoznawane przez *spasm* rozkazy obejmują (liczby w nawiasie oznaczają składnię rozkazu/grupy rozkazów - opisane dalej):

- rozkazy arytmetyczne (1):
  `add, addx, addcc, addxcc, sub, subx, subcc, subxcc`
- rozkazy logiczne (1):
  `and, andcc, or, orcc, xor, xorcc, andn, andcc, orn, orncc, xorn, xorncc`
- rozkazy przesunięć (1):
  `sll, srl, sra`
- rozkazy skoków warunkowych branch (2) (nawiasach `[ ]` nazwy alternatywne):
  `bn, bleu, ba, bgu, be [bz], bcs [blu], bne [bnz],`
  `bcc [bgeu], ble, bneg, bg, bpos, bl, bvs, bge, bvc`
- rozkazy odczytu(3) / zapisu(4) z pamięci:
  `ld, ldub, ldsb, lduh, ldsh, st, stb, sth, std`
- rozkazy sterujące:
  `call` (5), `jmpl` (6)
- inne rozkazy:
  `sethi` (7), `save` (1), `restore` (1)

Składnia rozkazów:

(1) *instr* <src1>, <src2>/<imm13>, <dst>
(2) *branch* <etykieta>
(3) *load* [<adres>], <dst>
(4) *store* <dst>, [<adres>]
(5) call <etykieta>
(6) jmpl <adres>, <dst>

(7) sethi <const>, <dst>

gdzie:

- <src1> - pierwszy rejestr źródłowy
- <src2> - drugi rejestr źródłowy
- <imm13> - stała 13 bitowa
- <dst> - rejestr docelowy wyniku
- <etykieta> - etykieta (adres) w sekcji kodu
- <const> - stała liczbowa (lub adres wskazywany przez etykietę danych/kodu)
- <adres> - kombinacja kilku powyższych składników:
  - <src1> + <src2>
  - <src1>
  - <src1> + <const>
  - <src1> - <const>
  - <const>

Dokładniejsze informacje dot. składni, formatu słów rozkazowych i kodowania rozkazów można znaleźć w specyfikacji SPARCv8 [1]. Tam również znajdują się rozkazy procesora SPARC, których implementacja **SmArc** nie obejmuje (inne rozkazy arytmetyczne, rozkazy korzystające z koprocesora, itp.)

Pseudorozkazy rozpoznawane przez *spasm* oraz ich interpretacja znajdują się w poniższej tabeli:

Pseudorozkaz	**Rozkaz(y)**	
mov %src, %dst	or	%g0, %src, %dst
cmp %reg1, %reg2	subcc	%reg2, %reg1, %g0
tst %reg	orcc	%g0, %reg, %g0
not %reg	xnor	%g0, %reg, %reg
neg %reg	sub	%g0, %reg, %reg
inc %reg	add	%reg, 1, %reg
dec %reg	sub	%reg, 1, %reg
clr %reg	or	%g0, %g0, %reg
restore	restore	%g0, %g0, %g0
save	save	%g0, %g0, %g0
ret	jmpl	%i7 + 8, %g0
jmp <etykieta>	jmpl	<etykieta>, %g0
skipz	bnz	. + 8 [1] [2]
skipnz	bz	. + 8 [1] [2]
set <const>, %reg	sethi	%hi(<const>), %reg
	or	%reg, %lo(<const>), %reg
		lub
	or	%reg, <const>, %reg
		(gdy <const> $\leqslant$ <imm13>)
nop	sethi	0, %g0

[1] Kropka (.) w polu argumentu oznacza adres bieżącego rozkazu;

[2] **SmArc** nie obsługuje *annul bit*, więc generowane rozkazy nie ustawiają tego bitu(!)

3.3. Dyrektywy SmArcAs

Postać dyrektywy	**Funkcja**
.text	Początek sekcji programu (rozkazów)
.data	Początek sekcji danych
.align k	Wyrównanie adresu do granicy k bajtów (k = 2 lub 4)
.skip n	Rezerwacja n kolejnych bajtów
.byte w1,w2,w3,...	Nadanie wartości w1, w2, .. kolejnym bajtom
.half w1,w2,w3,...	Nadanie wartości w1, w2, .. kolejnym półsłowom
.word w1,w2,w3,...	Nadanie wartości w1, w2, .. kolejnym słowom
.ascii 'znaki..'	Umieszczenie w kolejnych bajtach kodów znaków ASCII [3]
=	Przypisanie nazwie (lewostronnej) wartości (prawostronnej)

Inne założenia:

- małe i duże litery są rozróżniane za wyjątkiem nazw symboli specjalnych,
- rozkazy są pisane wyłącznie małymi literami,
- symbole specjalne zaczynają się od znaku procentu (%)
- etykiety i nazwy zaczynaja się od litery; po etykiecie musi być dwukropek (:).
- liczby mogą być w zapisie dziesiętnym lub szesnastkowym; w zapisie szesnastkowym liczby muszą być poprzedzone znakiem zera i litery 'x' (np. 0xffff),
- łańcuch znaków umieszczany jest w pojedynczym cudzysłowie ('),
- dostępne operatory unarne:
  - %lo(<const>) - wydziela 10 mniej znaczących bitów stałej (32-bitowej)
  - %hi(<const>) - wydziela 22 bardziej znaczących bity stałej (32-bitowej)
- nazwy rejestrów mogą być zapisywane w formie %r<N> lub alternatywnie jako %g<M>, %o<M>, %i<M>, %l<M>, %sp oraz %fp (gdzie $0 \leqslant N \leqslant 31$, $0 \leqslant M \leqslant 7$).

3.4. Format pliku wykonywalnego maszyny wirtualnej

Asembler *spasm* scala część danych oraz część kodu programu do pliku wynikowego o następującym formacie:

[3] Znaki sterujące zapisuje się następująco: \\$b$ - backspace, \\$n$ - new line, \\$r$ - carriage return, \\$t$ - horizontal tab.

Pole	Długość	Typ	Zawartość
`NAGŁÓWEK`	4 bajty	ciąg ASCII	stały ciąg '`SMARC`' identyfikujący plik wykonywalny
`WERSJA`	1 bajt	cyfra 0-9 w kodzie ASCII	numer wersji pliku wykonywalnego (obecnie jedyna obsługiwana wersja to '0')
`ID_DANYCH`	4 bajty	ciąg ASCII	stały ciąg '`DATA`' zapowiadający sekcję danych
`DL_DANYCH`	4 bajty	słowo 32-bitowe (big-endian)	zawiera długość bloku danych (w bajtach) wyrównanego do granicy słowa 32-bitowego
`DANE`	`DL_DANYCH` bajtów	sekcja danych	
`ID_KODU`	5 bajtów	ciąg ASCII	stały ciąg '`INSTR`' zapowiadający sekcję kodu
`DL_KODU`	4 bajty	słowo 32-bitowe (big-endian)	zawiera długość bloku kodu (w bajtach)
`KOD`	`DL_KODU` bajtów	sekcja kodu	

4. Instalacja

4.1. Wymagania

flex-devel, bison-devel, stdc++-devel, Gtk-devel, glade-devel, etc.

4.2. Budowanie pakietu SmArc

```
$ tar xzf smarc-XX.tar.gz
$ cd smarc
$ make
```

Bibliografia

[1] SPARCv8 (32-Bit SPARC) Architecture Book, `http://www.sparc.org/standards.html`