Architektury systemów komputerowych

Projekt 1: "Asembler dla architektury MIPS"

Przed przystąpieniem do implementacji projektu należy zapoznać się z treścią dodatkowe rozdziału Assemblers, Linkers, and the SPIM Simulator książki "*Computer Organization and Design: The Hardware/Software Interface"*.

Celem listy jest zbudowanie uproszczonego asemblera dla architektury MIPS, dzięki czemu dowiemy się jak działają ostatnie etapy kompilacji dla języków wyższego poziomu.

Kolejne punkty należy wykonywać po kolei. Język programowania użyty do implementacji jest dowolny (proszę wybrać taki, który ma odpowiedni zestaw narzędzi w bibliotece standardowej), ale asembler musi działać pod systemem Linux. Twój asembler powinen zakładać, że porządek bajtów jest określony według konwencji Little Endian. Kodowanie instrukcji ma być zgodne z MIPS ISA. W razie wątpliwości można podglądać co robi środowisko MARS. Zestaw instrukcji wraz z semantyką jest opisany w ściągawce MIPS Reference Data Card.

Uwaga: Należy starannie przemyśleć działanie swojego asemblera. Studenci którzy już realizowali ten projekt spędzali dużo czasu na poprawianie błędnie zaprojektowanej struktury programu. Dobra architektura to mniej kodu nad którym łatwiej panować. Zanim zaczniesz projektować swój asembler przeczytaj wszystkie wskazówki.

Wskazówki: Pierwszy przebieg asemblera tworzy reprezentację całego pliku wejściowego w postaci listy linii – rekordów: $(etykieta, instrukcja\ /\ dyrektywa, lista\ argumentów, komentarz)$. Następnie wykonujemy pierwszy przebieg kompilacji – będziemy tworzyć zawartość sekcji `.text` i `.data`, gdzie sekcja to zwyczajny kontener na ciąg bajtów. Oprócz tego będziemy przypisywać etykietom adresy absolutne i twórzyć listę relokacji – miejsc, które należy odwiedzić w drugim przejściu asemblera, aby obliczyć i wstawić referencje do etykiet.

1. **[5]** Zaimplementuj asembler tłumaczący instrukcje arytmetyczne procesora MIPS na kod maszynowy. Na wejściu Twój asembler ma pobierać plik tekstowy, gdzie każda linia będzie w nastepującym formacie:

   ```
   instrukcja    operand1,operand2,... # komentarz
   ```

   Komentarz jest opcjonalny, a ilość operandów będzie zależna od instrukcji. Na razie operandami będą wyłącznie rejestry: `$0`...`$31` i ich odpowiedniki symboliczne `$v0`, `$ra`, itd. oraz stałe całkowitoliczbowe. Instrukcje, które musi rozpoznawać asembler to: `lui`, `addi`, `addiu`, `slti`, `sltiu`, `andi`, `ori`, `xori`, `lui`, `sll`, `srl`, `sra`, `sllv`, `srlv`, `srav`, `mfhi`, `mthi`, `mflo`, `mtlo`, `mult`, `multu`, `div`, `divu`, `add`, `addu`, `sub`, `subu`, `and`, `or`, `xor`, `nor`, `slt`, `sltu`.

   Plik wyjściowy ma zawierać listing programu – tj. oryginalną zawartość programu, gdzie do każdej linii zawierającej instrukcję jest dołączony prefiks z zapisem szesnastkowym adresu instrukcji oraz słowa kodujące instrukcję, np.:

   ```
   00400000      02328020      add    $s0,$s1,$s2
   00400004      2268FFF4      addi   $t0,$s3,-12
   ```

   Wskazówki: Postaraj się wymyśleć swój asembler tak, by był łatwo rozszerzalny. Pisanie kodu do przetwarzania każdej instrukcji z osobna jest niemądrym pomysłem – lepiej jest wymysleć kodowanie metadanych dla każdej instrukcji np.: "`add:R:reg,reg,reg:0/%0/%1/%2/0/32`", "`addi:I:reg,reg,sint16:8/%1/%0/%2`". Osobno parsuj instrukcje, a osobno każdy z operandów.

2. **[2]** Doimplementuj możliwość stosowania dyrektyw – ich znaczenie jest objaśnione w podrozdziale A.10. Dyrektywy, które musi obsługiwać asembler to: `.align`, `.ascii`, `.asciiz`, `.byte`, `.data`, `.half`, `.space`, `.text`, `.word`. Odpowiednio zmodyfikuj listing. Niektóre z dyrektyw zmieniają bieżący adres docelowy asemblacji, a niektóre umieszczają dane pod bieżącym adresem asemblacji. Dyrektyw `.text` i

`.data` można używać wielokrotnie – wymaga to zmiany bieżącej sekcji do której dopisujemy bajty. Zauważ, że instrukcje zawsze muszą się znajdować pod adresami podzielnymi przez 4. Dyrektywy składowania danych będą wyglądać w listingu następująco:

```
10000000      68656c6c6f20776f726c640a00 .asciiz       “hello world!\n”
```

Wskazówki: Aby zachować powyższy format dla każdej linii wejściowej można utworzyć dodatkowy rekord o następującym formacie: (*adres*, *start w sekcji*, *ilość bajtów*) i utworzyć specjalną procedurę zajmującą się wyłącznie drukowaniem sekcji.

3. **[4]** Dodaj do swojego asemblera możliwość stosowania etykiet, czyli symbolicznego zapisu adresów. Następnie dodaj możliwość stosowania instrukcji skoków warunkowych i bezwarunkowych: `beq`, `bne`, `j`, `jal`, `jr`. Teraz każda linia pliku wejściowego może wyglądać jak jeden z poniższych wariantów:

```
etykieta1:    instrukcja    operand1,operand2,... # komentarz
etykieta2:    .dyrektywa    argument1,argument2       # komentarz
```

Zauważ, że od tego momentu operandem instrukcji lub dyrektywy `.word` może być również etykieta! Czasami adres instrukcji do której skaczemy może być zbyt daleko – w takim wypadku zgłoś błąd. Przyjmijmy, że etykieta to ciąg znaków opisany następującym wyrażeniem regularnym: `[A-Za-z][A-Za-z0-9_]*`.

Wskazówki: Zbuduj tablicę symboli, gdzie każdy element jest krotką: `(symbol, sekcja, offset)`. **Symbol** to nazwa etykiety, **sekcja** to `.text` lub `.data`, **offset** to adres relatywny względem początku sekcji. Twój asembler będzie musiał teraz zrobić dwa przebiegi – pierwszy zbierający informacje o etykietach, a drugi uzupełniający adresy w odpowiednich miejscach w binarnej reprezentacji programu.

4. **[2]** Zaimplementuj w swoim asemblerze obsługę następujących pseudoinstrukcji: `move`, `blt`, `bgt`, `li`, `la`. Niektóre z nich mogą być kodowane do więcej niż jednej instrukcji podstawowej, inne wymagają rozbicia adresu lub stałej symbolu na dwie 16-bitowe liczby. Pseudoinstrukcja `li` musi umieć akceptować 32-bitową stałą, a `la` absolutny adres etykiety.

Wskazówki: Podobnie jak w punkcie pierwszym, dobrze jest wymyśleć jakieś metadane służące do opisu pseudoinstrukcji np.: “`blt:P:reg,reg,label:slt $at,%0,%1;bne $at,$0,%2`”. Przed pierwszym przebiegem będzie trzeba uruchomić ekspander pseudoinstrukcji. Zauważ, że taka funkcjonalność jest zaczynem implementacji **makroasemblera**.

5. **[2]** Rozszerz swój asembler o możliwość stosowania instrukcji operujących na pamięci. Trudnością w tym punkcie jest dodanie operandów reprezentujących tryby adresowania: `($a0)` lub `offset($a0)`. Nowe instrukcje, które należy wspierać to: `lb`, `lbu`, `lh`, `lhu`, `lw`, `lwu`, `sb`, `sh`, `sw`.