

Konstruera och verifiera med SystemVerilog

DEL 2

Nya avancerade hårdvarubeskrivande högnivåspråk ger förhoppningar om en ökad produktivitet och förbättrad kvalitet. Det här den andra och avslutande delen av en artikel som använder Bindesh Patel, Novas Software och Stuart Sutherland, Sutherland HDL, Inc SystemVerilog som ett exempel på ett sådant språk.

Dynamiska processer används för att lägga till parallellism i en systemkonstruktion eller testbänksmodell. Målet är att öka effektiviteten via bättre resursanvändning. SystemVerilog utökar Verilog's "fork-join"-uttrycksblock med `join_any`- och `join_none`-modifierare. Operationen visas grafiskt i fig 11.

Eftersom processer delar resurser (t ex minne och data) är synkronisering ett måste. Detta kan göras med hjälp av semaforer och/eller events. Mailboxer används för datakommunikation mellan processer. Semaforer och brevlådor är inbyggda klasser, medan event är en utökad Verilog-datatyp.

Debug av parallella händelser är en utmaning eftersom tillståndsinformation måste behållas för varje process. Det ökar mängden lagrade data. Dessutom skapar samtidigheten en egen uppsättning felmöjligheter. Dessa inkluderar:

- Deadlock
- Fel beroende på data race
- Synkroniseringsfel

För att debugga sådana fel kan innehållet i en "named fork" göras synlig för användaren i interaktiv debug. Mera information kan sparas i traceminnet på ett sätt som liknar dokumenteringen av assertion-försök. Det kan till exempel gälla namnet på uppdelningen, när uppdelningen gjordes och associationer med andra kontextuella beroenden. Eftersom delade objekt bara kan nås via döpta objekt, till exempel brevlådor, semaforer och events, är det viktigt att övervaka dem under interaktiv debug eller slutsimulering för att identifiera race-fel eller synkroniseringsfel. Genom att använda assertions som övervakar

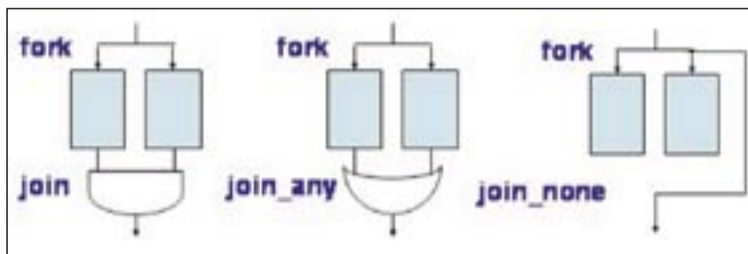


Fig 11. Block genererade av en "fork-join"-process.

tillgången till delade resurser via kommunikationskanalen kan man lättare hitta till exempel deadlock-problem.

ASSERTIONBASERAD KONSTRUKTION OCH DEBUG

Assertions används för att verifiera konstruktionen. De är väldefinierade, men ändå uttrycksfulla sätt att beskriva önskade eller oönskade beteenden hos ett block eller en grupp av block. Typiskt används assertions parallellt med en annan verifieringsmetod för att lokalisera valideringen i en "design and conquer"-ansats. Tanken är att verifiera en eller flera delar av konstruktionen och sedan kombinera resultaten för att se om helheten är giltig, dessutom används ett "scoreboard" för att identifiera vilka delar av konstruktionen som har verifierats med en viss grad av säkerhet och vilka som ännu inte har mött sina kriterier. Det här är vad som brukar kallas täckning.

Både täcknings- och assertion-beskrivningar kan användas samma underliggande temporala språk. I SystemVerilog kallas språket SVA (SystemVerilog Assertions). Direktivet på toppnivå kan vara `assert` eller "cover for asserting behaviour" eller inhämtning av täckningsdata.

Även om assertions huvudsakligen används för verifiering kan de också användas för konstruktion. De kan användas i syntes för att syntetisera ett block, det har varit framgångsrikt för gränssnitts-syntes. De kan också användas som virtuella komponenter för att förenkla distribuerad konstruktion och tidig högnivåsimulering. Sådana ansatser kan leda till en tidig förståelse av konstruktionspotentialen och dess begränsningar och förbättra produktiviteten för konstruktionsgruppen.

Assertions kan användas för att lokalisera och isolera fel om de placeras på gränssnitten. Gränssnitt i kombination med assertions är också ett kraftfullt sätt att kontrol-

lera om implementationen möter timing- och protokollkrav och för att göra prestandaanalys. I SystemVerilog kan assertions placeras inuti gränssnitt (kommunikation), eller i modeller (beteende).

Men att debugga assertions kan bli ett problem. I takt med att tilliten till assertions ökar blir de också allt mera komplexa och kräver debug. Ett alternativ är att använda vågformsdisplay. Källkodsannotation för objekt fungerar också bra för både konstruktion och assertions. Men att annotera assertion-kod är inte så lätt, framför allt beroende på två saker:

- Assertionkod är deklarativ, detta är en stor skillnad mot HDL-implementeringskod, som är procedurbaserad.
- Språket döljer en hel del. Till exempel döljer repetitionsoperatoren (*) i fig 12 många av de interna repetitionsstegen.

Det här kräver extra uppmärksamhet. På samma sätt måste också många av de andra sätten att visualisera utvidgas för att klara assertions.

BINDESH PATEL,
NOVAS SOFTWARE,
STUART SUTHERLAND,
SUTHERLAND HDL, INC

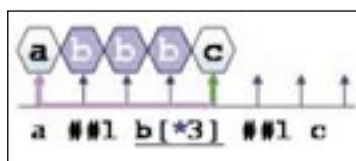


Fig 12. Repetitionsuttryck.