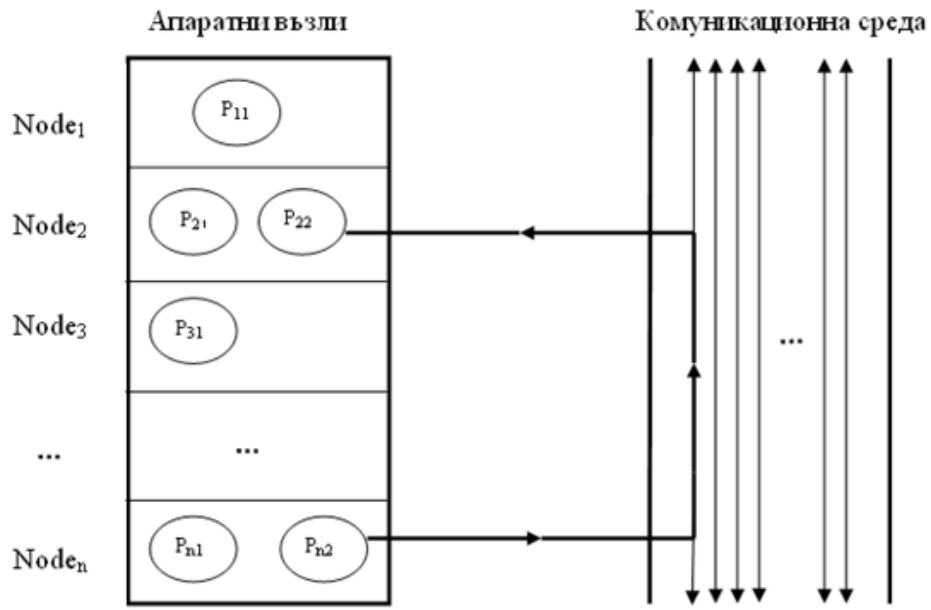
Един от основните математически модели за описание както на системи с обща, така и на системи с разпределена памет, е *CSP*.

CSP (*Communicating Sequential Processes*, Взаимодействащи последователни процеси) е предложен от Чарлз Хоар [16, 31]. Моделът се базира на понятието *процес* като базова активна единица. Процесът сам по себе си може да бъде съставен, т.е. да представлява съвкупност от други процеси. Структурата на всеки процес може да се определи с рекурсивния израз:

P : = < линеен участък > < комуникация > *P*

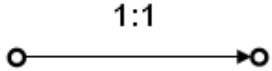
Следователно, в тялото на процеса съществуват два основни участъка – линеен участък, представляващ последователност от оператори и участък, в който процесът взаимодейства с останалите процеси в системата.



Фиг. 1.8. Структура на *CSP* машина

Изборът на тази структура на процесите е естествено следствие от разширяването на последователните машини и последователните езици за програмиране със средства за комуникация, при което се отчитат особеностите на разпределения модел памет. В съответствие с този модел се определя абстрактна *CSP* машина, която се състои от множество апаратни компютърни възли[1] и комуникационна среда за връзка между възлите (фиг. 1.8).

Всеки един процес се изпълнява в локална защитена среда (апаратния възел). Единственият начин за взаимодействие с останалите процеси е обменът на съобщения по свързващите ги комуникационни канали.



Фиг. 1.9. Двуточково еднопосочно взаимодействие

Обменът на съобщения, т.е. взаимодействието, се извършва чрез комуникационната среда, която представлява множество физически канали. Всеки такъв канал служи за двуточково еднопосочно взаимодействие (фиг. 1.9).

Комуникацията е с двустранна синхронизация от типа *рандеву* (*rendezvous*). Това означава, че и двата процеса трябва да достигнат командата за взаимодействие, за да може то да се осъществи. Процесът, който е изпреварил другия се задържа (*блокира*[2]) толкова време колкото е необходимо. Подобен избор се обосновава с необходимостта да се използват максимално опростени канали - еднопосочни, без буферизация и без контрол на потока (flow control).

Дефинирани са няколко оператора на *CSP*, които се разглеждат и като команди.

Операторът за *паралелна композиция* (паралелната команда) има следния синтаксис

$P = \{ P_1 \,||\, P_2 \,||\, \ldots \,||\, P_n \}$

Паралелната команда определя съставния процес *P* като съвкупност от *n* на брой паралелни процеса, които се стартират и изпълняват едновременно. Командата приключва с приключването на най-продължителния от съставящите я процеси.

---

1 *Апаратният възел* (*Node*) е завършен компютър - със собствен процесор, собствена памет и интерфейсни схеми за връзка със средата.

2 Даден процес се разглежда като *блокиран*, ако е изведен от опашката на диспечера на операционната система (опашката на готовите задачи). Механизмът на блокирането на процесите служи за намаляване на системните загуби - избягват се циклите на изчакване, които товарят непроизводително процесора.

Операторът за *последователна композиция* (последователната команда) има следния синтаксис

$P = P_1; P_2; \ldots; P_n$

Последователната команда служи за конструиране на съставния процес *P*, състоящ се от *n* последователно изпълнявани процеса.

Операторът за *алтернативен избор* има следния синтаксис

$P = \{G_1 \rightarrow P_1 \square G_2 \rightarrow P_2 \square \ldots \square G_n \rightarrow P_n\}$

Този оператор е подобен на класическия оператор за избор *if/else*. Но за разлика от него, поддържа недетерминизъм, откъдето е известен и като оператор за *недетерминиран избор*. Недетерминизмът е вътрешно присъщ на всяка една паралелна система, което фактически е отчетено с въвеждането на оператора за алтернативен избор.

Двойката $G_i \rightarrow P_i$ се нарича *i-та* алтернатива, където защитата (*guard*) G_i се разглежда като булева променлива[3], а P_i е подчинения на G_i процес. P_i може да се изпълни единствено, ако защитата му G_i е истина.

Изпълнението на оператора за алтернативен избор протича по следната схема: *Едновременно* се проверяват всички защити. Ако нито една от защитите не е истина, операторът се прекратява. Ако само една от защитите G_i е истина, се изпълнява нейния подчинен процес P_i. Ако повече от една защита е истина, по *случаен закон* се избира една от тях и се преминава към изпълнение на подчинения й процес. Ключови са едновременната, а не последователната, проверка на защитите и случайният, а не детерминираният, закон за избор.

Алтернативната команда фактически е обобщение и съдържа като частен случай оператора за детерминиран избор *if/else* от последователните езици.

Операторът

* { < *тяло* >}

определя *цикличен процес*. За неговото управление се налага да се използват двата елементарни процеса *STOP* и *SKIP*. При достигане на *STOP* командата се прекратява, докато при *SKIP* се извършва преход в началото й.

Командите за *взаимодействие* са две – за изпращане на съобщение и за приемане на съобщение.

Командата за изпращане (send) на съобщението *msg* по канала *c* има вида

---

3 Най-простият вариант на *защита* (guard) е булева променлива. Често обаче се среща и значително по-сложния вариант, когато ролята на защита изпълнява *команда за взаимодействие*. Затова е редно защитата да се разглежда като предикат, който връща булев резултат.

c ! *msg*

Съответно, командата за получаване (receive) на съобщението от канала *c* и записването му в променливата *var* е от вида

c ? *var*

И двете команди за взаимодействие използват директно именуване на каналите и променливите. Това дава възможност на всеки един процес да определи момента на взаимодействието и процеса, с който се взаимодейства.

Важна особеност на *CSP* е, че той представлява едновременно математически модел, формална спецификация на паралелна система и език за паралелно програмиране. Нещо повече, съществуват реални апаратни системи, отговарящи на *CSP* машината и реални езици за паралелно програмиране, производни на *CSP* (фиг. 1.10).

Фиг. 1.10. Връзка между абстрактната и реалната страна на *CSP*

Първият пример за практическо приложение на *CSP* са транспютърните паралелни системи и езикът за паралелно програмиране *OCCAM* [5, 14, 40].

Транспютрите (TRANSistor comPUTER) представляват специализирани едночипови микрокомпютри, предназначени за изграждане на масово-паралелни процесори (*MPP*). За разлика от универсалните процесори, например *Intel x86*, те осигуряват цялостна апаратна поддръжка на процеса, като базов активен

обект, и превръщането му в елементарна единица работа[4]. Освен това притежават и вградени комуникационни средства – канали. Произвеждани са от английската фирма *InMOS*.

През 2005 е създадена фирмата *XMOS*. Един от основателите ѝ - Дейвид Мей (David May) е водещия архитект на транспютрите. Фирмата *XMOS* предлага в момента микропроцесорната фамилия *XS1*, която в своята основа е много близка до транспютрите и е следващата реализация на *CSP* машината.

Device	XS1-L1	XS1_L2	XS1-G2	XS1-G4
XCores	1	2	2	4
Threads	8	16	16	32
MIPS (max)	400/500	800	800	1600
SRAM (total)	64 KBytes	128 KBytes	128 KBytes	256 KBytes
OTP (total)	8 KBytes	16 KBytes	16 KBytes	32 KBytes
I/O	3v3	3v3	3v3 (5v tolerant)	3v3 (5v tolerant)
Power consumption	15-200mW	30-400mW	200-700mW	200-1200mW
Packages (I/O)	QFP64 (36) QFP128 (64)	QFN124 (84)	BGA144 (88)	BGA144 (88) BGA512 (256)

Фиг. 1.11. Процесори от фамилия *XS1*

На фиг. 1.11 са показани основните параметри на някои микропроцесори от тази фамилия: устройствата с ниска консумация *XS1-L1*, *XS1-L2* и високопроизводителните *XS1-G2*, *XS1-G4*. Фамилията непрекъснато се развива и допълва с нови представители. Актуална информация може да се получи от сайта на фирмата *XMOS*[5].

И четирите посочени разновидности на фамилията *XS1* съдържат едно или повече ядра от типа *XCORE*. Това ядро представлява събитиен (*event-driven*) *RISC* процесор със силносвързан *I/O*. Поддържа едновременното изпълнение на *8* нишки[6], което го причислява към *SMT/TLP* процесорите (фиг. 1.12). Използва се *дребногранулирано многонишково* изпълнение (*fine-grained SMT*, *Simultaneous Multi-threading*). Във всеки машинен цикъл се извършва превключване на контекст на нишка, за разлика от *едрогранулираното многонишково* изпълнение (*coarse-grained SMT*) при *Intel x86* и *PowerPC*. Парале-

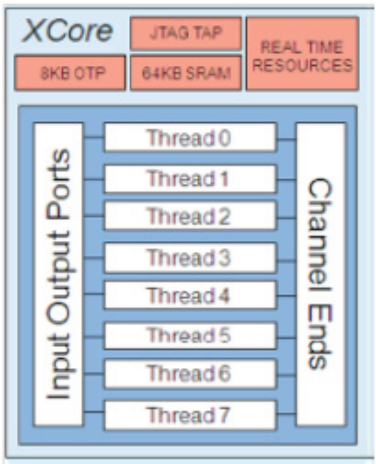
4 Критерий за тежестта на дадена работа за процесора е отношението на времето за нейното изпълнение към времето за изпълнение на основните машинни инструкции (зареждане/запомняне, аритметико-логически и др.). Елементарна е такава единица, за която това отношение е в диапазона от 1 до не повече от 10.

5 Сайт на фирмата *XMOS*: www.xmos.com

6 Разглеждането ни не изисква разграничаването на понятията *процес*, *задача* и *нишка*.

лизмът на ниво нишки (*TLP*) и на ниво данни (*DLP*), синхронизацията и взаимодействието на процесите се поддържат изцяло апаратно на микроархитектурно ниво и не изискват ОС.

На фиг. 1.12 са показани основните ресурси, влизащи в състава на ядрото *XCORE*: *SMT/TLP* апаратна среда, поддържаща едновременното изпълнение на 8 нишки; силносвързаната входно-изходна система; физическите канали за връзка; оперативна памет от типа *SRAM*; постоянна (*OTP*) памет; ресурси за работа в реално време; *JTAG* интерфейс за връзка с инструменталния компютър.



Фиг. 1.12. Ресурси на ядрото *XCORE*

Фамилията *XS1* се програмира на езика *XC* [49, 51, 53]. *XC* е разширение на езика *C* с конструкции за паралелно програмиране – явно управление на паралелизма, взаимодействието, събитията и вход-изхода. Благодарение на апаратната поддръжка на *CSP* модела, операторите на *XC* се транслират в къси *RISC* инструкции, като се избягват служебните загуби, характерни за ОС.

Както е показано на фиг. 1.10, *XC* е реализация на *CSP*. При анализа на езика може да се проследи семантическата връзка между конструкцията *par* на *XC* и паралелната команда на *CSP*; между оператора *select* и алтернативната команда; между операторите за въвеждане *:>* и извеждане *<:* и командите за изпращане *!* и получаване *?* на съобщения и т.н.

Определено може да се каже, че *XC*, като реализация на *CSP*, в голяма степен отговаря на изискването на Чарлз Хоар, автора на *CSP*: „*the highest goal of programming language design is to enable good ideas to be elegantly expressed*".